Université de Montréal

MODÉLISATION D'UN RÉSEAU INTÉGRÉ SUR PUCE BASÉ SUR UNE ARCHITECTURE EN ANNEAU

François Deslauriers
Département de Génie informatique
École Polytechnique de Montréal

Mémoire présenté en vue de l'obtention du
Diplôme de Maîtrise ès Sciences Appliquées
(Génie électrique)

Août 2005

Your file Votre référence
ISBN: 978-0-494-16871-4
Our file Notre référence
ISBN: 978-0-494-16871-4

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MODÉLISATION D'UN RÉSEAU INTÉGRÉ SUR PUCE BASÉ SUR UNE ARCHITECTURE EN ANNEAU

présenté par : DESLAURIERS François
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

MME NICOLESCU Gabriela, Doct., présidente
M. BOIS Guy, Ph.D., membre et directeur de recherche
M. SAVARIA Yvon, Ph.D., membre et codirecteur de recherche
M. ABOULHAMID El Mostapha, Ph.D., membre

Remerciements

Je tiens d'abord à remercier mon directeur de recherche, Guy Bois. Son soutien moral constant m'a permis de mener à terme ce projet. Par ailleurs, je voudrais également remercier mon co-directeur de recherche, Yvon Savaria, de même que Michel Langevin pour l'apport d'idées et de pistes à exploiter qu'ils ont pu me fournir tout au long du projet.

Je ne pourrais pas passer sous silence l'immense contribution de mes confrères de travail : Bruno Lavigueur, Olivier Benny, Luc Filion, David Quinn, Mortimer Hubin, Jean-François Thibeault, Simon Provost, Patrick Samson et Francis « The Tank » St-Pierre. Grâce à eux, la maîtrise a parfois pu prendre la forme d'une récréation.

Mes parents sont également du nombre des gens à qui je dois une fière chandelle pour m'avoir encouragé dans mes démarches. Il en est de même pour ma copine Laurence et mon frère Yann.

Un beau merci par ailleurs à Réjean Lepage et Alexandre Vesey pour leur soutien technique, de même qu'à notre chère Ghyslaine qui est une mère pour nous.

Finalement, je me dois de profiter de l'occasion qui m'est offerte pour me remercier moi-même d'avoir persévéré tout au long de cette aventure. Je dois une partie de cette persévérance aux contributions immenses de mes fidèles amis Ghyslain Raza, Jonathan Zerg, Gregory Superi et Gary Broslma.

Résumé

La constante miniaturisation du transistor a révolutionné le marché des systèmes embarqués des dernières années. Grâce aux progrès réalisés dans le domaine des semi-conducteurs et aux possibilités offertes par les nouvelles technologies, la porte a été ouverte aux systèmes multiprocesseurs, systèmes qui permettent à plusieurs unités de se répartir les tâches à effectuer de façon à atteindre une vitesse de traitement bien supérieure aux systèmes utilisant un seul processeur.

Une des conditions essentielles pour atteindre de tels gains de performance est de diminuer autant que possible le temps de communication entre les composants. En effet, la synchronisation entre diverses unités de traitement implique en général une communication constante entre ces derniers. De plus, l'intégration d'une centaine de blocs sur une même puce engendre une imposante quantité de transactions. L'architecture de communication utilisée se doit d'être en mesure de supporter cette hausse des communications.

Les architectures de communication dominant le marché actuel sont celles utilisant les bus, que ce soit dans leur forme la plus simple ou dans une version hiérarchisée. Or, ces architectures de bus seront prochainement déficientes puisqu'elles ne permettent d'accomoder qu'un nombre limité de blocs (moins de cinq processeurs, par exemple). Ceci est nettement insuffisant pour les systèmes intégrés sur puce du futur qui comprendont des centaines de composants.

Pour contourner ce problème, un nouveau concept a fait son apparition depuis quelques années : le réseau intégré sur puce. L'utilisation de ce type d'architecture permet de rencontrer les besoins en bande passante et en extensibilité.

Ce travail met d'abord en lumière les déficiences des architectures de bus en mettant l'accent sur leurs carences les plus importantes. Ensuite il présente le concept de réseau pur puce. Ce type d'architecture incorpore des notions générales associées aux réseaux à grande échelle. Il est donc primordial de présenter quelles sont ces notions générales qui seront ensuite intégrées au monde des systèmes sur puce.

Il existe par ailleurs plusieurs façons d'agencer entre eux les composants qui forment le réseau. On parle ici de la notion de *topologie*. Plusieurs topologies sont étudiées et caractérisées selon leurs avantages et inconvénients. Une dizaine de réseaux intégrés sur puce, issus de travaux universitaires ou commerciaux sont d'ailleurs présentés pour valider la théorie sur les topologies.

Une nouvelle architecture de réseau intégré, basée sur le modèle de l'anneau par jeton (en anglais *Token Ring*) et baptisée RoC (*Rotator on Chip*), est présentée et détaillée. Cette architecture facilement extensible se veut un réseau qui maximise le taux d'utilisation de ses composants, ce qui le rend très peu coûteux en espace tout en permettant d'obtenir de bonnes performances. Les simulations montrent qu'il est plus lent que les architectures en mailles, mais qu'il necéssite un espace plus restreint sur la puce. En ayant en tête le perpétuel compromis performance/coût, le RoC se démarque particulièrement lorsque le concepteur doit se soucier de la surface occupée par le réseau en même temps que de la puissance qu'il dissipe. Le RoC se démarque également pour les applications necéssitant un trafic local et il est un choix avantageux lorsqu'utilisé pour traiter les applications de type *streaming*.

Abstract

In recent years, technology scaling caused the embedded systems' world to change drastically. Because of constant progress and interesting outcomes connected to new technologies, multiprocessor systems have emerged as the ultimate solution to process many embedded applications such as multimedia and telecommunications.

The expected performance gains can be reached only to one condition: communication time between components must be lowered as much as possible. Indeed, synchronisation between processing units involves high and constant communications. Moreover, integrating hundreds of IP cores on a single chip leads to a significant increase in the number of transactions. The communication infrastructure must then be able to sustain the required bandwidth and latency.

Currently, bus-based architectures are very popular, whether simple shared based or advanced hierarchical structures. Nevertheless, typically, these architectures can only support up to 5 processors and up to twelve masters before becoming the system bottleneck. This is obviously insufficient for systems on chip integrating over a hundred cores, which will be widely used in the near future.

To solve this problem, a new paradigm has emerged since the year 2000: the network on chip approach. Such networks easily allows meeting bandwidth needs and scalability requirements.

This work highlights bus-based architecture deficiencies by showing why they are not suitable for multiprocessor SoCs and then presents the network on chip concept. This type of architecture uses general notions associated with wide area networks. Those concepts are thus presented in this work

In addition, it is possible to connect network components together in many ways, leading to various topologies. Several of those topologies are presented in this work and are discussed in order to highlight their pros and cons. Some NoC architectures are then presented to go along with the discussions.

A new NoC architecture, based on the token ring model and called RoC (*Rotator on Chip*) is the main focus of this thesis. Its architecture is easily scalable and can support a high utilization rate, which makes it less expensive in area than other networks while still preserving acceptable performance. Simulations show that RoC is slower than mesh-based networks on chip, while being less expensive. Having in mind the performance/cost tradeoff, RoC is very suitable when area and power consumption are significant issues. RoC also provides very good performance when used to process stream-based applications.

Table des matières

Liste des figures

Liste des tableaux

Liste des acronymes

ACK	**A**cknowledge
AHB	**A**dvanced **H**igh-speed **B**us
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture
APB	**A**dvanced **P**eripheral **B**us
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuits
AVI	**A**udio **V**ideo **I**nterleaved
BCA	**B**us **C**ycle **A**ccurate
BE	**B**est **E**ffort
DCT	**D**iscrete **C**osine **T**ransform
DMA	**D**irect **M**emory **A**ccess
DNS	**D**omain **N**ame **S**ystem
ECLIPSE	**E**mbeddeb **C**hip-**L**evel **I**ntegrated **P**arallel **S**upercomputer
FIFO	**F**irst **I**n **F**irst **O**ut
flit	**fl**ow control dig**it**
GALS	**G**lobally **A**synchronous **L**ocally **S**ynchronous
GT	**G**uaranteed **T**raffic
HyRoC	**H**yper **R**ing **o**n **C**hip
IP	**I**ntellectual **P**ropery
ISO	**I**nternational **S**tandards **O**rganisation
Ko	**K**ilo **o**ctects
LAN	**L**ocal **A**rea **N**etwork
MOPS	**M**illion **O**perations **P**er **S**econd
MP3	**M**otion **P**icture Experts Group Audio Layer **3**
MPEG4	**M**otion **P**icture **E**xperts **G**roup - **4**
MPSoC	**M**ulti-**P**rocessor **S**ystem **o**n **C**hip
NACK	Non **Ack**nowledge

NI	**N**etwork **I**nterface
NoC	**N**etwork **o**n **C**hip
NoCGEN	**N**etwork **o**n **C**hip **Gen**erator
OPB	**O**n-chip **P**eripheral **B**us
OCP	**O**pen **C**ore **P**rotocol
OSI	**O**pen **S**ystems **I**nterconnection
PA	**P**in **A**ccurate
PCI	**P**eripheral **C**omponent **I**nterconnect
PerNoC	**Per**formance **N**etwork **o**n **C**hip
PLB	**P**rocessor **L**ocal **B**us
RAM	**R**andom **A**ccess **M**emory
RoC	**R**otator **o**n **C**hip
RTOS	**R**eal **T**ime **O**perating **S**ystem
SAD	**S**um of **A**bsolute **D**ifference
SoC	**S**ystem **o**n **C**hip
SoCIN	**S**ystem **o**n **C**hip **I**nterconnection **N**etwork
SOCP	**S**ystemC **O**pen **C**ore **P**rotocol
SPIN	**S**calable **P**rogrammable **I**nterconnection **N**etwork
StEPNP	**S**ystem-**L**evel **E**xploration **P**latform for **N**etwork **P**rocessors
TDMA	**T**ime **D**ivision **M**ultiple **A**ccess
TF	**T**imed **F**unctional
TOPS	**T**era **O**perations **P**er **S**econd
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
UCT	**U**nité **C**entrale de **T**raitement
UTF	**U**n**T**imed **F**unctional
VCI	**V**irtual **C**omponent **I**nterface
VHDL	**V**HSIC **H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
VSIA	**V**irtual **S**ocket **I**nterface **A**lliance

Liste des annexes

Introduction

Depuis une quinzaine d'années, les grands progrès technologiques dans le domaine des systèmes intégrés sur puce (SoC, pour *System on Chip*) ont fait considérablement croître les possibilités. Il n'y a pas si longtemps, un processeur pouvait occuper à lui seul un dé (puce). Les communications avec les mémoires et autres périphériques se faisaient via l'utilisation d'une carte (board). Chacun des composants du système conçu était une puce dont l'ensemble constituait la carte.

Or, les progrès réalisés au niveau transistor permettent aujourd'hui d'inclure plusieurs blocs fonctionnels sur une seule et même puce. Avec la poursuite de cette évolution, les transistors étant de plus en plus petits et de plus en plus rapides, il sera possible d'ici quelques années de concevoir un SoC comportant l'équivalent d'un milliard de portes logiques pouvant opérer à des fréquences avoisinant la dizaine de giga hertz (GHz) [ITRS04]. Cela ouvre la porte à l'intégration de fonctionnalités multiples dans un même système. Les champs d'opération sont immenses. Que ce soit dans le domaine de la sécurité des systèmes (caméras de surveillance), de la santé (prothèses auditives), des communications (téléphonie cellulaire), du divertissement (lecteurs MP3) ou de la photographie (caméras numériques), les SoC sont tout désignés pour remplir les exigences du marché en termes de performance et de coût [HEWC04]. L'avènement récent des SoC apporte donc une solution intéressante aux besoins propres de ces domaines. Ils offrent la possibilité d'intégrer un ensemble de processeurs et de coprocesseurs spécialisés pour exécuter les différentes tâches du système.

Les SoC du futur seront basés sur le traitement de signaux numériques, avec des charges allant de 10 MOPS pour du traitement audio jusqu'à 1 TOPS pour la génération synthétique de vidéo. Pour accomplir ce traitement, un seul processeur ne suffira pas à la tâche, d'où l'apparition prochaine des systèmes sur puce multiprocesseurs (MPSoC) [BEBE04].

Problématique

Sur un système monoprocesseur, la fonctionnalité d'un système est décortiquée puis séparée en tâches spécifiques et relativement indépendantes entre elles. Ces tâches peuvent être ordonnancées sur l'unité centrale de traitement (UCT) par un système d'exploitation temps réel (RTOS), par exemple. Elles s'exécuteront concurremment sur l'UCT, laissant croire à un parallélisme pourtant absent.

Il est possible d'ajouter du parallélisme avec l'introduction de modules matériels dédiés qui permettent, selon la tâche à effectuer, d'obtenir des gains substantiels sur la performance comparativement à une même tâche effectuée par une UCT (de l'ordre d'une centaine de fois plus rapide). D'ailleurs, un intéressant domaine de recherche est le partitionnement logiciel/matériel des modules de façon à obtenir un compromis optimal entre performances et coûts [CBRB04].

Qui plus est, la naissance des MPSoC apporte obligatoirement avec elle le concept de parallélisme dans les opérations. Plusieurs UCT peuvent se partager une portion d'exécution de tâche (par exemple, l'encodage d'une image) ou exécuter plusieurs tâches indépendantes (par exemple, un système de surveillance par caméra). Tel que mentionné plus tôt, la technologie actuelle permet d'intégrer plusieurs de ces processeurs, coprocesseurs et modules matériels. Le nombre d'opérations s'exécutant en même temps se trouve alors augmenté de façon drastique.

D'un autre côté, l'augmentation du nombre d'opérations implique aussi l'augmentation des communications sur la puce. En effet, un processeur a parfois besoin d'obtenir une information se trouvant sur une mémoire externe, de demander à un coprocesseur d'effectuer une opération spécifique, etc. Ces communications entraînent un mouvement de données d'un composant à un autre. La multiplication des initiateurs de telles transactions apporte donc la multiplication des données circulant sur la puce. Selon

[BEBE04], les systèmes de demain seront dominés par les communications plutôt que par le traitement.

Actuellement, les architectures de communication sont assez rudimentaires, bien qu'elles soient tout à fait adéquates pour les besoins actuels. Les architectures de bus [IBM03] [ARM01] [SILI02] [SONI01] permettent d'acheminer une transaction ayant été initiée par un module maître (ex : processeur) vers un module esclave (ex : mémoire). Toutefois, lorsqu'un système comporte plus d'une dizaine de modules maître, les communications deviennent le goulot d'étranglement du système et ne permettent plus de profiter du parallélisme, puisque plusieurs modules peuvent attendre le résultat des transactions qu'ils ont initiées [BEDE02].

Depuis quelques années, un nouveau concept a fait son apparition dans le design d'un SoC : les réseaux intégrés sur puce (NoC pour *Network on Chip*). Les NoC procurent une amélioration significative aux communications en permettant à plusieurs transactions de s'effectuer en même temps, à l'instar des grands réseaux informatiques tels que le réseau Internet.

Objectifs

Le principal objectif de ce mémoire est de concevoir un modèle de NoC et d'évaluer ses performances par rapport à des topologies connues. En effet, plusieurs recherches ont déjà été entamées dans ce domaine, recherches ayant conduit à l'élaboration de plusieurs configurations possibles pour obtenir un NoC. Par contre, il n'existe toujours pas de consensus à ce sujet et l'on en est encore à l'exploration architecturale de même qu'à l'évaluation de la contribution réelle d'un réseau dans un SoC. Par conséquent, un second objectif se veut une réflexion sur l'utilisation des NoC selon leurs caractéristiques et selon le type d'application.

Méthodologie

Afin de répondre aux objectifs fixés, plusieurs tâches devront être réalisées. Une première étape est l'exploration des opportunités offertes par la plate-forme StepNP, développée en langage SystemC [OSCI03] par la société StMicroelectronics. Cette plate-forme de haut niveau renferme une bibliothèque de blocs de propriété intellectuelle tels que processeurs, mémoires et réseaux d'interconnexions [PAPB02]. Une architecture en développement peut être facilement modifiable par l'ajout ou le changement d'un composant par un autre, ce qui facilite les comparaisons et les prises de décision. De plus, tous les outils essentiels au développement d'une architecture à haut niveau (simulateur, débogueur et différents mécanismes d'analyse de performance) sont fournis et leur utilisation est relativement simple. Enfin, le code source de la plate-forme nous était disponible et gratuit, ce qui est très utile dans le contexte académique où les travaux ont été réalisés.

Une deuxième étape sera de construire le modèle de base du réseau. Deux niveaux d'abstraction seront explorés: UTF (*untimed functional*) et TF (*timed functional*), selon la méthodologie de co-design. Le niveau UTF permet de ne pas tenir compte des notions de temps pour permettre de se concentrer sur la fonctionnalité du système, c'est-à-dire ce qu'il doit accomplir. Le niveau TF ajoute du *timing*.

Une troisième étape sera d'apporter plusieurs améliorations à l'algorithme de base, d'où en découlera plusieurs variantes du modèle initial. L'analyse de ces variantes donnera des informations sur leur utilisation dans un contexte donné et selon le type d'application. Une application complète préalablement développée, MPEG4 (*Motion Picture Experts Group 4*), servira de banc d'essai.

Originalité et contribution

Une analyse récente des NoC existants, effectuée par [HEWC04], suggérait l'exploration du concept de hiérarchisation dans les réseaux, pour maximiser les avantages offerts par la localisation des ressources. Il est de plus en plus important d'exploiter cette localité

dans le contexte des SoC où le temps de propagation des signaux devient une source de problèmes pour les concepteurs. Les réseaux sur puce modélisés ici tentent de remédier à ce problème et invitent les concepteurs à profiter des avantages de la localité des ressources lorsque vient le temps de faire l'assignation des composants sur la puce.

Par ailleurs, ce travail se veut un apport important à l'exploitation d'une plate-forme multiprocesseurs, puisqu'il offre au concepteur d'une telle plate-forme différentes configurations possibles du réseau dans le but de satisfaire le plus possible à ses besoins en termes de bande passante et de latence.

Finalement, ce réseau sur puce tente de pousser au maximum la réutilisation de ses composantes ainsi que leur taux d'utilisation afin de donner une justification solide pour l'espace supplémentaire que le réseau occupe sur la puce ainsi que la puissance qu'il consomme, tout ceci dans un contexte très contraignant pour les développeurs de systèmes intégrés sur puce.

Distribution des chapitres

Ce mémoire est constitué de cinq chapitres. Le chapitre 1 survole le monde des systèmes intégrés sur puce, en mettant l'accent sur les limitations des architectures actuelles de communication. Le chapitre 2 introduit une idée qui se veut une solution prometteuse aux problèmes de communication : les réseaux intégrés sur puce. Une partie de ce chapitre est aussi consacrée à l'explication de différents concepts que l'on peut retrouver dans les réseaux à grande échelle. En effet, plusieurs similarités existent entre ces types de réseaux et les réseaux intégrés sur puce. Le chapitre 3 décrit ensuite les composants du modèle de base du réseau développé alors que le chapitre 4 regroupe tous les ajouts apportés à ce modèle de même que les améliorations qu'ils procurent au niveau des performances du réseau. Puis, les résultats sont présentés au chapitre 5. Enfin, pour clore ce mémoire, la conclusion résume le travail accompli et renferme également une courte discussion portant sur les travaux futurs et autres considérations.

CHAPITRE 1 Revue de littérature

1.1. L'état actuel

Les systèmes, les applications et la technologie sont en constante évolution. Cette situation donne lieu à de nouveaux besoins. Afin de mieux cerner ces besoins, il est important de bien comprendre dans quelle portion de l'évolution nous nous retrouvons.

1.1.1. L'écart de productivité

Un des enjeux actuels des systèmes sur puce est que la technologie évolue trop rapidement par rapport aux capacités des développeurs d'en tirer profit.

Gordon Moore, co-fondateur d'Intel et de Fairchild, annonça en 1965 que la densité des transistors intégrés sur une puce doublerait tous les 18 mois. Cette prédiction, éventuellement nommée « Loi de Moore », est encore validée de nos jours et est illustrée à la Figure 1.1.

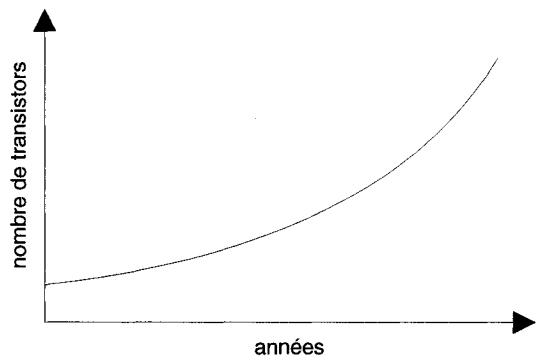


Figure 1.1 : Allure de la loi de Moore

D'un côté, cette importante croissance de la densité des transistors permet aux ingénieurs de créer des systèmes sur puce toujours plus denses, plus complexes et plus rapides

qu'auparavant. D'un autre côté, la productivité d'un concepteur augmente également avec le temps, mais de façon beaucoup plus limitée. L'amélioration des outils de développement et de synthèse se fait lentement par rapport à la vitesse à laquelle la technologie évolue. Le fossé qui se creuse entre technologie et productivité est représenté par la Figure 1.2.

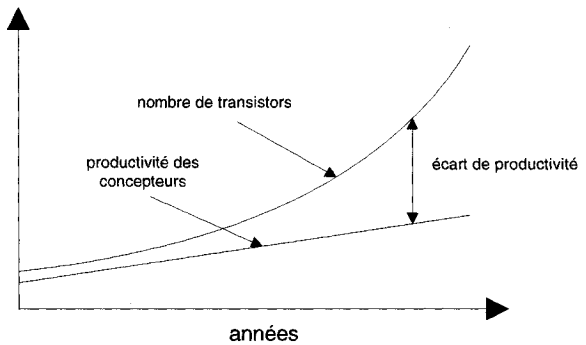


Figure 1.2 : L'écart de productivité

Par ailleurs, la loi de Moore devrait encore s'appliquer pour une période variant entre cinq et dix ans. Par conséquent, pour la prochaine décennie à venir, les concepteurs de systèmes devront adapter les méthodologies de conception de façon à leur permettre d'exploiter le maximum des ressources qui seront mises à leur disposition.

1.1.2. Raffinement progressif

Il existe plusieurs techniques qui tentent de réduire l'écart de productivité. Une première technique, appelée *raffinement progressif*, consiste à développer sur plusieurs niveaux successifs d'abstraction [FILI02]. On appelle *niveau d'abstraction* le fait de négliger des spécificités de l'implantation pour se concentrer sur une partie du problème. Par exemple, au niveau UTF aucune notion de temps n'est considérée et seuls les fonctionnalités du système de même que les algorithmes sont validés. Lorsque ce niveau de développement est complété, on passe au niveau d'abstraction suivant, nommé TF, où une certaine notion

de temps est introduite (l'ordre des opérations est important et une synchronisation entre les modules doit être présentée).

Par la suite, on passe au niveau BCA (*bus cycle accurate*) où les communications entre modules sont décrites abstraitement sous forme de transactions. Cela conduit au dernier niveau PA (*pin accurate*) qui représente l'évolution des signaux proprement dits dans le temps. Le niveau PA est celui qui exprime le niveau le plus élevé de détail avant la synthèse elle-même. La Figure 1.3 récapitule les différentes étapes de la méthodologie.

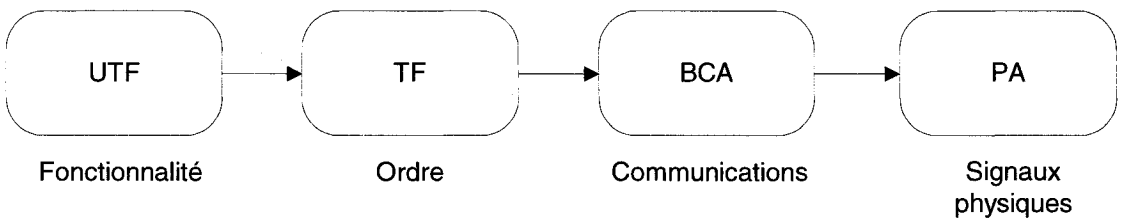


Figure 1.3 : Le raffinement progressif

Plusieurs compagnies, comme par exemple Synopsys [SYNO03], tentent de développer des outils et des langages qui couvrent toutes les étapes pour accélérer et automatiser la conception jusqu'à la synthèse du système, de façon à réduire son temps de développement.

1.1.3. Réutilisation

Une deuxième technique, appelée *réutilisation*, permet d'intégrer les fonctionnalités d'un module déjà existant sans avoir à le concevoir de nouveau. Les développeurs se sont rapidement aperçus qu'il était impossible de construire un système entier (comme par exemple un encodeur MPEG ou un routeur sophistiqué) à partir de rien. Les systèmes sont maintenant si complexes qu'il est très ardu d'obtenir les résultats escomptés en partant d'une plate-forme complètement générique. Également, de plus en plus de pression est mise sur l'ingénieur afin de réussir à concevoir ces systèmes dans un délai

raisonnable. En effet, les échéanciers pour demeurer concurrentiel dans le marché sont de plus en plus courts (parfois moins d'une année).

La réutilisation des composants est apparue comme étant une solution bénéfique dans le processus de conception. Les compagnies productrices de systèmes ont commencé à se doter d'une bibliothèque de composants réutilisables, au premier rang desquels se trouvent les simples portes logiques. Plus tard, des blocs relativement primitifs tels qu'additionneurs, compteurs et multiplexeurs se sont joints aux entités réutilisables. De nos jours, des processeurs entiers, des mémoires et d'autres modules spécialisés et complexes sont réutilisés dans la conception puisqu'il n'est pas avantageux de les redéfinir. Bon an mal an, les sociétés qui concoivent des puces ont accès à des centaines de blocs IP (*intellectual property (modules)*) pour élaborer leur système global. Dans une vision naïve du processus de conception, les concepteurs n'ont qu'à intégrer les blocs requis en les faisant interagir entre eux adéquatement à travers des interfaces communes et par des communications appropriées. Cette portion du travail représente tout de même un défi important au niveau de la standardisation des communications. Ce sujet sera approfondi plus loin.

La réutilisation est donc une technique omniprésente dans la réalisation de systèmes sur puce et continuera de l'être. Dans le chapitre suivant, le concept de réseau intégré sur puce sera décrit et on notera que les avantages qu'offrent la réutilisation ne seront pas mis de côté.

1.1.4. Applications du présent et du futur

Tel que mentionné plus tôt, les champs d'application des SoC sont très vastes. De plus en plus de systèmes électroniques peuvent être intégrés sur une seule et même puce. Ces systèmes peuvent regrouper plusieurs fonctionnalités souvent indépendantes les unes des autres. On n'a qu'à penser aux téléphones cellulaires qui permettent également la

navigation sur Internet, l'envoi de messages texte, la prise de photos numériques et la capture de séquences vidéo.

On considère deux types d'applications : les applications orientées *données* ainsi que les applications orientées *contrôle*. Les applications orientées données manipulent un grand flot d'information. Par exemple, une caméra vidéo numérique échantillonne plusieurs fois par seconde des images de l'environnement filmé pour ensuite les convertir en pixels (points de couleur) pouvant être affichés simultanément sur un écran. Les données traversent plusieurs blocs d'un même système où elles subissent différents traitements (ex. : conversion en pixels, correction gamma, amplification, encodage, etc.).

Les applications orientées contrôle, comme le nom l'indique, doivent contrôler un environnement. Par exemple, un simple thermostat permet d'ajuster la température ambiante à un niveau précis. Un système orienté contrôle est souvent composé de senseurs qui sont responsables de prendre des mesures de l'environnement contrôlé (ex. : la température d'une pièce). Ces résultats de mesure sont alors envoyés au système exploitation qui vérifie si une action doit être apportée à l'environnement.

Dans ces deux types d'application, les communications sont généralement nombreuses mais leurs caractéristiques diffèrent. Une application orientée données enverra une grande quantité d'information et nécessitera donc une grande bande passante. Une application orientée contrôle générera beaucoup moins de trafic; par contre, ce trafic devra être acheminé le plus rapidement possible puisque le contrôle de l'environnement en dépend.

1.2. Limitations des systèmes sur puce actuels

Cette section se veut une présentation des défis qui attendent la prochaine génération de systèmes sur puce. Les requis de performance y sont présentés, de même que les moyens à prendre pour atteindre ces nouveaux standards.

1.2.1. Systèmes sur puce

Les systèmes sur puce fournissent des solutions intégrées aux problèmes de design rencontrés dans les domaines des télécommunications, du multimédia et autres domaines grands consommateurs d'électroniques [BEDE02]. Le monde des SoC est en pleine effervescence et surtout en continuelle progression. Dans quelques années, des systèmes comprenant quatre milliards de transistors opérant à une fréquence de dix GHz seront monnaie courante. Actuellement, plusieurs modules IP remplissant différentes fonctions et opérant à différentes fréquences doivent être intégrés sur une même puce, ce qui augmente le défi de conception [GPIS03]. Les ressources disponibles sur une puce étant immenses, un simple processeur ne peut plus utiliser l'ensemble des transistors du dé. Il est donc possible de fractionner la puce en différentes régions ayant des fonctionnalités distinctes (voir Figure 1.4) [KJSF02].

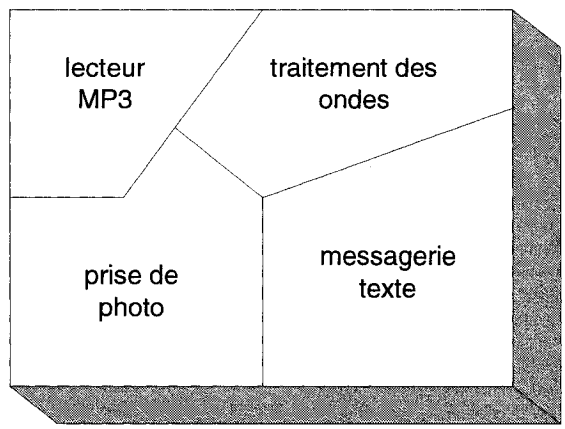


Figure 1.4 : Fractionnement de la puce en systèmes hétérogènes

Une nouvelle façon de concevoir le système peut alors être envisagée. Notamment l'approche GALS (*Globally Asynchronous, Locally Synchronous*) permet à plusieurs régions de la puce d'opérer à des fréquences différentes (il est aussi possible qu'elles

opèrent à la même fréquence mais dans des domaines de phase différents; le sujet sera approfondi plus loin).

1.2.2. Premier problème : les fils

Il n'y a pas si longtemps, il était possible de négliger le temps de propagation d'un signal sur un fil sans que cette négligence n'entraîne une erreur catastrophique de conception. Le délai de propagation des fils était largement inférieur aux délais d'un transistor ou d'une porte logique. Avec la largeur du canal d'un transistor qui sera bientôt de 50 nm, la fréquence d'opération de ce transistor pourra atteindre 10 GHz. C'est donc dire que deux fronts montants d'horloge seront séparés par 100 picosecondes seulement. La vitesse de propagation de la tension étant limité à une fraction de celle de la lumière, le signal a le temps de parcourir moins de 30 mm avant que le prochain front montant ne survienne! Dans une puce où les fils sont très entremêlés, un signal pourrait prendre entre six et dix périodes d'horloge pour se propager sur la puce entière [GPIS03b]. Le problème est déjà rencontré dans les designs actuels, où le temps de propagation d'un signal s'étend sur une période complète d'horloge. Même si les longs délais peuvent être contrôlés par des techniques de pipelinage, l'incertitude apportée par ces délais sera tout de même considérable pour les développeurs [BEDE02b]. L'ajout de répéteurs pour garder le délai linéaire plutôt que quadratique est donc requis et il est difficile de placer proprement ces répéteurs, d'autant plus qu'ils s'ajoutent aux contraintes d'espace et de puissance déjà présentes [DATO01].

Par ailleurs, la taille réduite des transistors permet d'inclure plus de blocs sur la puce, ce qui engendre forcément plus de fils pour relier les blocs en question. Du point de vue global, plus de fils sont sollicités à la fois, ce qui cause une augmentation sensible de la puissance dissipée [BEDE02]. Aussi, des fils ne suivant aucune structure ont une capacité parasite difficile à prévoir tôt dans le processus de conception. Ils engendrent également des interférences (en anglais : *crosstalk*) envers les fils adjacents, ce qui rend le comportement hautement imprévisible [DATO01].

De même, un filage *ad hoc* présente souvent la caractéristique qu'une importante proportion des fils ne sont utilisés que 10% du temps ou moins, ce qui n'est pas souhaitable dans le mesure où les fils prennent une importance grandissante dans le système [WIGO02]. Finalement, la miniaturisation fait en sorte que les transistors nécessitent maintenant des tensions d'opération de plus en plus faible pour éviter qu'ils ne claquent. Prochainement, il ne sera pas rare d'observer des tensions inférieures à 1V. L'intégrité du signal est donc menacée, puisqu'une légère variation de la tension peut maintenant faire la différence entre un 1 logique et un 0 logique. Somme toutes, les fils deviennent un enjeu majeur dans le développement de systèmes sur puce. Il est à noter que les impacts des problèmes mentionnés plus haut deviennent encore plus grands au fur et à mesure que la technologie évolue.

1.2.3. Deuxième problème : la synchronisation

Les enjeux entourant les fils ont des conséquences directes sur la synchronisation du circuit. Puisqu'il sera bientôt impossible qu'un signal se propage d'une extrémité de la puce à l'autre en une période d'horloge, la synchronisation globale de tous les blocs IP perdra sa signification. Il sera donc important de limiter la distance parcourue par les signaux critiques de façon à garantir la performance du système global [BEBE02].

Une solution intéressante est donc de fragmenter le système en plusieurs sous-systèmes distincts. Ainsi, tous les blocs d'un même sous-système seraient synchronisés entre eux sur le même signal d'horloge. Par contre, les sous-systèmes n'auraient pas cette synchronisation et devraient se synchroniser par les communications. Le réseau d'interconnexions devrait donc être en mesure d'assumer cette synchronisation. Voilà pourquoi cette approche, présentée plus tôt comme l'approche *GALS*, devient attrayante : les ingénieurs peuvent se séparer la tâche en travaillant parallèlement sur des sous-systèmes différents sans se soucier du détail fin de la synchronisation avec les autres. Les communications devraient en principe ramener tout les signaux au même niveau lorsque

vient le temps des échanges d'informations. Ces communications deviendront omniprésentes au fur et à mesure que le nombre de modules sur une puce continuera d'augmenter.

1.2.4. Troisième problème : les communications

La possibilité d'inclure plusieurs dizaines d'unités de traitement sur une puce représente une solution intéressante pour un ingénieur toujours avide de concevoir des systèmes performants et rapides. Le parallélisme dans les opérations est une façon rêvée d'accomplir une tâche le plus rapidement possible. Toutefois, le fait d'exécuter cette tâche avec *N* processeurs ne garantit pas que le traitement se fera *N* fois plus rapidement, surtout si le degré de dépendance est grand entre les processeurs. On peut faire l'analogie avec un groupe d'ingénieurs qui travaillent sur un système commun. Bien que chaque ingénieur puisse travailler sur une portion du système, chacun doit être au courant des travaux exécutés par les autres, via des réunions et des rencontres de mise à jour. L'ingénieur ne passera donc pas tout son temps à faire progresser son travail; la communication prendra une grande part de son temps. Il en sera de même dans les futurs systèmes sur puce multiprocesseurs. Un modèle souvent cité dans les systèmes futurs est composé de processeurs esclaves faisant « ce qu'on leur demande » et de maître(s) répartissant le travail à effectuer aux autres unités de traitement. Ceci engendre une importante communication entre tous ces modules qui doivent communiquer entre eux. À cela s'ajoute la communication que l'on retrouve sur les SoC actuels, comme par exemple les échanges traditionnels de données entre maîtres et esclaves (écritures et lectures en mémoire). Les communications deviendront donc rapidement le goulot d'étranglement de la prochaine génération des systèmes sur puce, c'est donc ce qui en limitera les performances.

1.3. Les architectures de bus

Présentement, les communications sont assurées par des architectures de bus sur puce qui sont une adaptation des bus sur carte que l'on retrouve sur des ordinateurs personnels, le bus *PCI* par exemple. La prochaine section présente un survol des architectures de bus existantes ainsi que leurs limitations.

1.3.1. AMBA

Depuis 1999, ARM [ARM01] propose le protocole AMBA en tant que norme pour les systèmes sur puce. La spécification offre deux protocoles qui peuvent être utilisés selon les besoins.

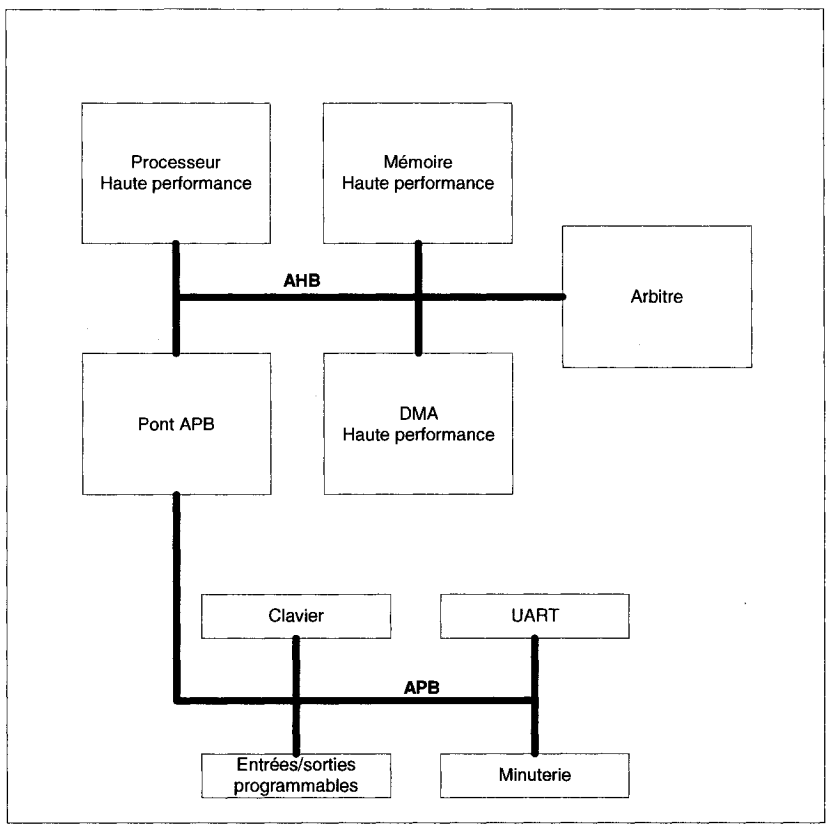


Figure 1.5 : Structure du bus AMBA

Tout d'abord, le *Advanced High-speed Bus* est adapté pour les communications à haute vitesse requises par les modules performants du système comme les processeurs, par exemple. Ensuite, le *Advanced Peripheral Bus* est utilisé pour connecter les modules qui n'ont pas besoin de la performance offerte par AHB. Les périphériques comme les minuteries et les contrôleurs d'entrées/sorties sont tout désignés pour ce genre de protocole. Un pont APB permet de relier des bus gérés selon les deux protocoles. Selon les spécifications de AMBA, ce pont n'a comme utilité que de fournir une interface plus simple. Toute latence présentée par un périphérique de basse performance sera reflétée par le pont APB au bus AHB. Le pont ne peut agir comme maître qu'au niveau APB, où il sera d'ailleurs le seul maître. Un exemple de système utilisant le protocole AMBA est illustré à la Figure 1.5. Le lecteur est invité à consulter [BERT03] pour plus de détails sur les spécifications du protocole AMBA.

1.3.2. CoreConnect

CoreConnect [IBM03] est un protocole conçu par IBM et ayant plusieurs similarités avec AMBA. Le protocole est illustré à la Figure 1.6. On retrouve premièrement le *Processor Local Bus* pour les modules à haute performance tels que les processeurs, mémoires cache et DMA (*Direct Memory Access*). Il est à noter qu'une interface à un bus externe peut être ajoutée à ce niveau.

Le deuxième niveau, appelé *On-Chip Peripheral Bus* (OPB), est un bus secondaire dont le but principal est d'augmenter la performance du bus principal en diminuant la charge capacitive sur le PLB. Les périphériques comme les ports série, ports parallèles, UART (*Universal Asynchronous Receiver-Transmitter*), minuteries et autres modules de faible performance peuvent adéquatement être branchés à ce bus. Plusieurs maîtres sont supportés à ce niveau. Un pont fait également le lien entre le niveau PLB et OPB. Il est à noter que ce pont agit comme maître au niveau OPB et comme esclave au niveau PLB, contrairement à AMBA.

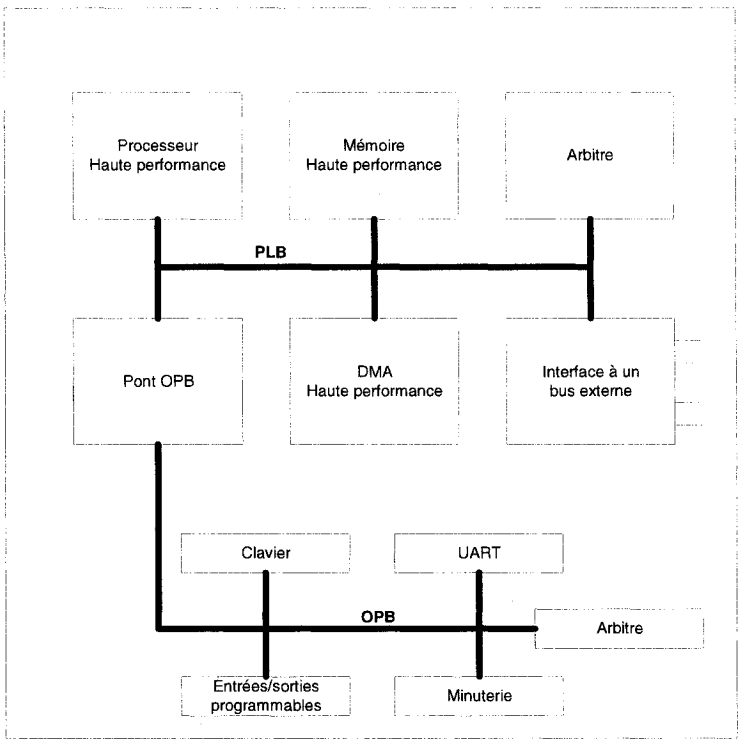


Figure 1.6 : Structure du bus CoreConnect

CoreConnect est un protocole de bus complet qui peut très bien faire parti d'un système de haute performance tel qu'une station de travail. Il supporte une lecture concurrente à une écriture, de même que des transactions de type *différé (split)*, où le maître laisse le contrôle du bus après avoir effectué sa requête pour le reprendre lorsque la réponse sera prête. Le désavantage de CoreConnect est qu'il est possiblement trop compliqué, en ce sens qu'il offre trop de fonctionnalités qui ne seront pas utilisées dans une simple application embarquée [USSE01]. Près d'une quarantaine de sociétés et d'organismes ont une licence de CoreConnect, dont l'École Polytechnique de Montréal.

1.3.3. SilliconBackplane III

SiliconBackplane III est un système d'interconnexions commercialisé par Sonics [SONI01] pour les applications multimédia. Ce système est implanté un peu différemment des architectures de bus AMBA et CoreConnect. Une représentation est

illustrée à la Figure 1.7. Chacun des composants est relié au réseau via un agent [WING01]. L'arbitration est basée selon un système rotatif à priorité qui découle du TDMA (*Time Division Multiple Access*). À chaque coup d'horloge, l'arbitre sélectionne l'agent correspondant à la période de temps en cours. Si l'agent n'a aucun message à envoyer, l'intervalle de temps est alloué à un autre agent selon l'algorithme *round-robin*.

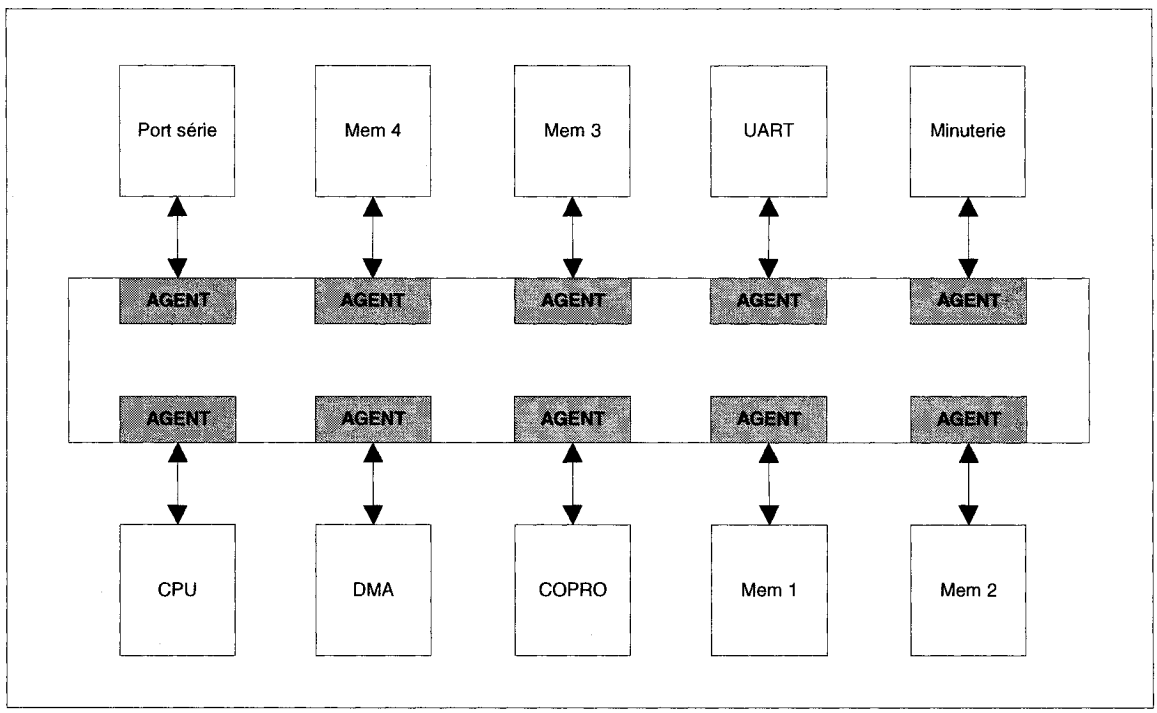


Figure 1.7 : Structure du bus SiliconBackplane

1.3.4. Wishbone

Wishbone [SILI02], développée par Silicore, est une spécification qui propose de réduire au minimum l'ensemble des signaux de façon à permettre des accès simples via un bus. Il n'y a donc pas de notion de bus opérant à différentes fréquences ni de ponts pour les relier (voir Figure 1.8). Si le concepteur souhaite avoir un système avec deux niveaux de bus, il est possible de le faire en créant deux interfaces *Wishbone*, plutôt qu'en définissant deux interfaces différentes (PLB et OPB par exemple). Par contre, le développeur doit

définir lui-même les sous-standards de Wishbone, comme par exemple l'ordre des données (*Little endian* versus *Big endian*). Aussi, des fonctionnalités supplémentaires pourraient devoir être ajoutées si nécessaires puisque non présentes dans la version de base de Wishbone.

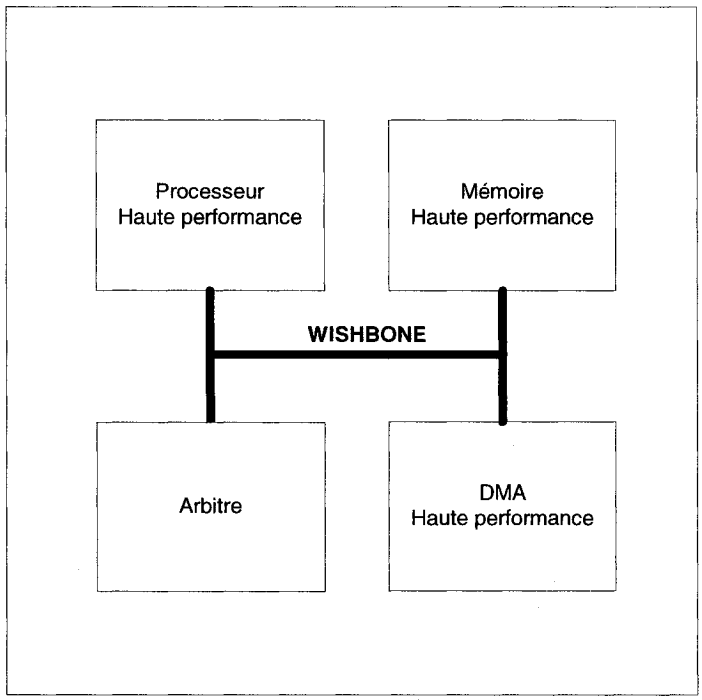


Figure 1.8 : Structure du bus Wishbone

1.3.5. Limitations des bus

Dans un bus partagé, lorsque plusieurs requêtes sont générées simultanément par plusieurs maîtres, l'arbitre décide de donner l'accès à un maître en particulier selon des règles d'arbitration. Un problème significatif de telles architectures est une dégradation de la performance causée par le nombre excessif de conflits. Une solution, telle qu'adoptée par AMBA et CoreConnect, entre autres, est d'instaurer une hiérarchie de bus contenant chacun les modules communiquant fréquemment ensemble, autant que possible. Cette solution est présentée à la Figure 1.9. Deux requêtes sur un bus partagé

(Figure 1.9a) produisent un conflit qui peut être évité par l'utilisation d'un bus hiérarchique (Figure 1.9b).

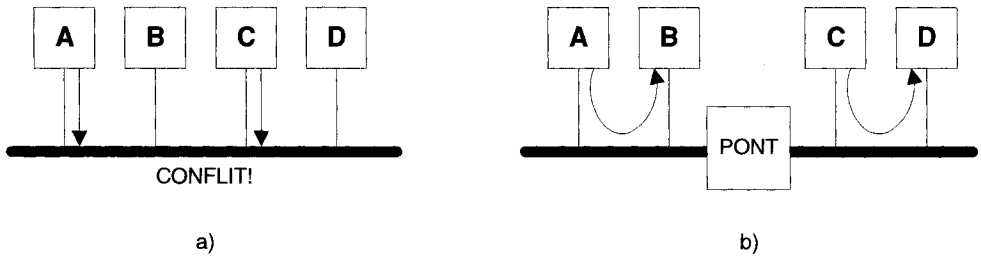


Figure 1.9 : Situation sur un bus partagé menant à un conflit réglé par l'utilisation d'un bus hiérarchique

Si on examine les considérations physiques, on observe une contradiction qui rend problématique l'utilisation des bus comme médium de communication dans un système complexe. D'un autre côté, plus il y a de modules connectés à un segment de bus, plus ce bus devra opérer à une fréquence d'horloge petite, ceci étant dû à la charge capacitive présente sur le bus. Pour garder une performance acceptable, il faut donc réduire le nombre de modules connectés à un même segment et par conséquent augmenter le nombre de segments. D'un autre côté, plus il y a de segments, plus la performance (latence) s'en trouve affectée à cause de la surcharge qu'apporte chaque pont reliant deux segments [YOO03]. Le bus le plus lent déterminera la performance des communications inter-segments. La solution pour diminuer la surcharge est donc de diminuer le nombre de segments, ce qui est contradictoire avec le problème précédent.

Par ailleurs, dans le cas où l'arbitration est faite selon la priorité de la requête, une requête de faible priorité pourrait prendre un temps considérable à être traitée si les requêtes de plus haute priorité se multiplient, ce qui peut entraîner un problème de famine. Il serait souhaitable, si on parle de qualité de service, de garantir une certaine bande passante à ce type de module. Toutefois, la bande passante du bus n'augmente pas avec le nombre de modules connectés à celui-ci, contrairement aux réseaux

conventionnels où la bande passante augmente lorsqu'on y insère un nœud supplémentaire [GPIS03b]. On dira alors du bus qu'il n'est pas extensible (en anglais *scalable*), ce qui est son problème majeur puisqu'au-delà d'un seuil limite, il est incapable de recevoir une charge supplémentaire. Finalement, les architectures de bus ne sont pas en mesure de répondre aux contraintes temps réel associées aux applications de réseautique, de télécommunications et de multimédia, puisqu'il est presque impossible de déterminer avec précision leur pire temps de réponse [WING01].

Néanmoins, les architectures de bus restent tout de même adéquates pour les SoC actuels qui intègrent moins de cinq processeurs et tout au plus une dizaine de maîtres [BEDE02]. Cependant, ces architectures seront inappropriées pour les futurs systèmes qui comprendront des centaines d'unités pouvant générer de l'information à être transférée. Avec des IP opérant à des fréquences de l'ordre du GHz, un seul bus (ou même plusieurs bus synchronisés) ne sera pas une solution viable, en raison des charges capacitives et de la résistance des fils qui ralentissent la propagation du signal [MAMA04].

Comme nous le verrons dans ce travail, il y a toutefois de l'espoir en ce qui a trait aux perspectives d'utilisation des bus dans les SoC du futur. En effet, dans l'architecture que nous proposons, le bus standard occupera encore une place de choix dans les communications.

1.3.6. Défis

Suite à la problématique énoncée dans la section précédente, voici une liste des principaux défis qui attendent les concepteurs dans le domaine des communications sur puce et des SoC en général :

- Fournir un réseau d'interconnexions dont les composants assureront des transmissions fonctionnelles et fiables. On peut y parvenir en exploitant des infrastructures et des protocoles déjà existants dans le domaine des réseaux à

grande échelle. Les interconnexions formeraient maintenant un micro-réseau sur puce, ce qui constituerait une adaptation du traditionnel modèle OSI.

- Parvenir au compromis flexibilité versus efficacité énergétique. Le système se doit de faire une gestion intelligence de la puissance dissipée. En effet, l'énergie résultante des communications globales ne diminue pas avec les nouvelles technologies, contrairement à l'énergie résultante du traitement. Cela s'explique par le fait que plus un transistor est petit, moins il consomme de puissance. Par contre, un fil ne change pas sensiblement d'une technologie à l'autre. L'énergie devient donc de plus en plus dominante dans les communications [BOZZ04].
- Fournir une qualité de service avec un budget limité en consommation d'énergie et en espace, le tout en tenant compte des limitations de la technologie. Une qualité se service inclut la performance et la fiabilité, sans se limiter à ces métriques. La performance est nécessaire pour répondre aux applications toujours plus exigeantes tandis que la fiabilité découle plutôt de la dépendance des consommateurs face à l'utilisation qu'ils font des accessoires électroniques dans la vie de tous les jours [BEDE02b].
- Étendre la réutilisation aux réseaux d'interconnexion pour permettre au concepteur d'utiliser les composants du réseau sur plusieurs systèmes [DATO01]. La réutilisation, si importante dans la méthodologie de conception des SoC, peut être facilitée par l'emploi d'interfaces communes qui attacheraient chacunes des composants au réseau.
- Permettre l'extensibilité des communications, ce qu'une architecture de bus n'est pas en mesure de faire. Il serait souhaitable que le réseau en place supporte l'ajout d'un nouveau bloc sans altérer les performances pour les blocs déjà en place. Ceci serait donc un bénéfice découlant directement de la réutilisation du réseau.

Ces défis sont intéressants pour les ingénieurs qui auront comme contraintes supplémentaires des échéanciers à respecter pour la mise en marché de nouvelles technologies en plus des problèmes d'intégrité de signal et une surface de puce limitée.

CHAPITRE 2 Les réseaux intégrés sur puce

Depuis quelques années, un nouveau concept a émergé dans le domaine des systèmes sur puce. Ce concept vise, essentiellement, à améliorer considérablement les communications dans les systèmes. L'idée est de reproduire sur une puce les infrastructures de communication que l'on retrouve dans les grands réseaux d'ordinateurs, comme les LAN et Internet. On parlera donc de réseaux intégrés sur puce (en anglais NoC pour *Network on Chip*). Cette section décortique plusieurs concepts réseau et NoC en présentant leurs caractéristiques ainsi que le contexte dans lequel ils s'inscrivent.

2.1.1. Modèle OSI

Pour éviter la multiplication des solutions d'interconnexion d'architectures hétérogènes, l'ISO (*International Standards Organisation*) a développé un modèle de référence appelé modèle OSI (*Open Systems Interconnection*). Ce modèle décrit les concepts utilisés et la démarche suivie pour normaliser l'interconnexion de systèmes ouverts (un réseau est composé de systèmes ouverts lorsque la modification, l'adjonction ou la suppression d'un de ces systèmes ne modifie pas le comportement global du réseau). Le modèle OSI n'est pas une véritable architecture de réseau, car il ne précise pas réellement les services et les protocoles à utiliser pour chaque couche. Il décrit plutôt ce que doivent faire les couches. Cette norme a été adoptée depuis une vingtaine d'années. Le modèle OSI comporte sept couches (voir Figure 2.1). Les couches basses (1, 2, 3 et 4) sont nécessaires à l'acheminement des informations entre les extrémités concernées et dépendent du support physique. Les couches hautes (5, 6 et 7) sont responsables du traitement de l'information relative à la gestion des échanges entre systèmes informatiques. La suite de cette section résume le rôle des diverses couches.

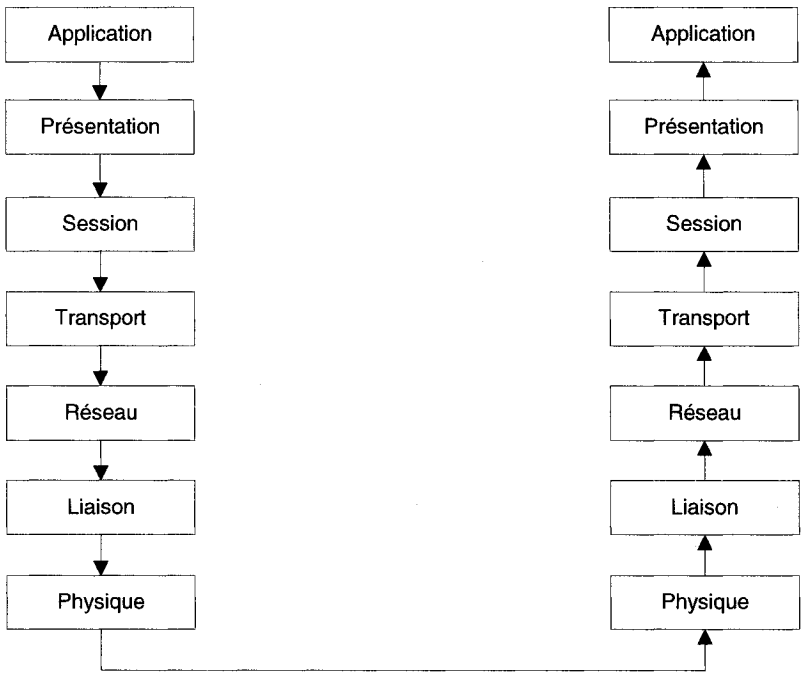


Figure 2.1 : Modèle OSI

2.1.1.1. La couche physique

La couche physique s'occupe de la transmission des bits de façon brute sur un canal de communication. L'unité d'information typique de cette couche est le bit, représenté par une différence de potentiel.

2.1.1.2. La couche liaison

Elle va transformer les signaux reçus de la couche physique en une liaison a priori exempte d'erreurs pour la couche réseau. Elle fractionne les données d'entrée de l'émetteur en trames, transmet ces trames en séquence et gère les trames d'acquittement renvoyées par le récepteur. De manière générale, un rôle important de cette couche est la détection et la correction d'erreurs apparues sur la couche physique. Cette couche intègre également une fonction de contrôle de flux pour éviter l'engorgement du récepteur. L'unité d'information de la couche liaison de données est la trame.

2.1.1.3. La couche réseau

C'est la couche qui permet de gérer le routage des paquets et l'interconnexion des différents sous-réseaux entre eux. La couche réseau contrôle également l'engorgement du sous-réseau. L'unité d'information de la couche réseau est le *paquet*.

2.1.1.4. La couche transport

Cette couche est responsable du bon acheminement des messages complets au destinataire. Le rôle principal de la couche transport est de prendre les messages de la couche session, de les découper s'il le faut en unités plus petites et de les passer à la couche réseau, tout en s'assurant que les morceaux arrivent correctement de l'autre côté. Cette couche effectue donc aussi le réassemblage du message à la réception des morceaux. Cette couche est également responsable du type de service à fournir aux utilisateurs du réseau : service en mode connecté ou non, avec ou sans garantie d'ordre de délivrance, diffusion du message à plusieurs destinataires à la fois, etc. L'unité d'information de la couche réseau est le *message*.

2.1.1.5. La couche session

Cette couche organise et synchronise les échanges entre des tâches distribuées. Elle établit également une liaison entre deux programmes d'application devant coopérer et commande leur dialogue (qui doit parler, qui parle...).

2.1.1.6. La couche présentation

Cette couche s'intéresse à la syntaxe et à la sémantique des données transmises : c'est elle qui traite l'information de manière à la rendre compatible entre les tâches communicantes.

2.1.1.7. La couche application

Cette couche est le point de contact entre l'utilisateur et le réseau. C'est donc elle qui va apporter à l'utilisateur les services de base offerts par le réseau, comme par exemple le transfert de fichiers, la messagerie, etc.

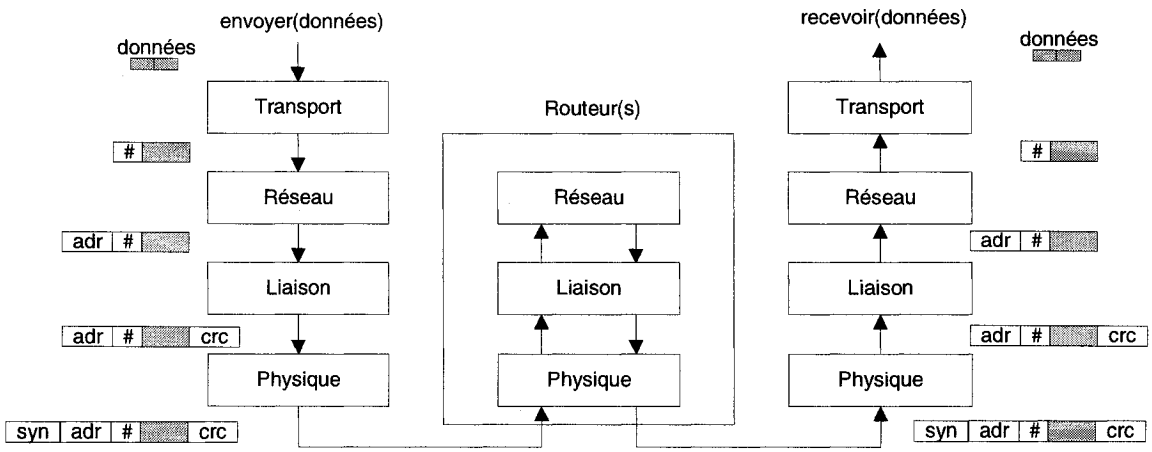


Figure 2.2 : Modèle OSI appliqué aux réseaux intégrés sur puce

2.1.1.8. Couches couvertes par un NoC

Pour une communication sur puce, les quatre dernières couches du modèle sont supportées puisque ce sont ces couches qui sont responsables de traiter les échanges d'informations entre deux composants du système (Figure 2.2).

2.1.2. Caractéristiques / Catégories de NOC

On peut dire des réseaux intégrés sur puce qu'ils représentent une approche prometteuse pour remplacer les bus. Tel que mentionné précédemment, un des principaux avantages est la réduction des effets électromagnétiques en introduisant une interconnexion structurée dans le but d'éliminer autant que possible les longs fils [SIBR04]. Un autre avantage est qu'une approche de communication en couches permet d'isoler les

implémentations physiques des couches de plus haut niveau comme les transactions, ce qui permet d'obtenir une meilleure bande passante avec moins de fils [MAMA04]. De plus, de telles architectures peuvent supporter un nombre pratiquement illimité de composants, ce qui les rend très extensibles, comparativement aux bus. Une fois que le protocole réseau est bien défini et standardisé, les composants du réseau peuvent être conçus, fabriqués et testés indépendamment avant d'être connectés ensemble. Du point de vue SoC, cela implique que des sous-systèmes indépendants peuvent potentiellement être développés séparément de la spécification avant d'être intégrés avec les autres sous-systèmes. Par ailleurs, il serait souhaitable, pour les SoC de demain, d'avoir des modules de traitement qui pourraient être employés dans différentes plates-formes selon un style de conception *plug-and-play*. C'est pour cette raison qu'un réseau sur puce modulaire et extensible représente une bien meilleure infrastructure de communication qu'un bus partagé [BEBE04]. Les réseaux seront donc préférés aux bus (une forme dégénérée de réseau) puisqu'ils offrent une plus grande bande passante et qu'ils supportent normalement plusieurs transactions simultanées [DATO01].

Un NoC peut être décrit par sa topologie et par les stratégies utilisées pour effectuer des opérations comme le routage, le contrôle de flot, la commutation, l'arbitration et le stockage d'informations [ZESU03]. La *topologie* peut être définie comme étant la façon dont sont répartis les canaux de communication et les nœuds qui s'y rattachent. Le *routage* détermine quel est le chemin qui sera suivi par les paquets pour passer d'une source A à une destination B, tandis que le *contrôle de flot* est responsable de l'allocation des canaux et des tampons à un message au fur et à mesure qu'il chemine dans le réseau. La *commutation* est le mécanisme qui retire un message d'un canal d'entrée d'un routeur pour le remettre sur un canal de sortie de ce même routeur. *L'arbitration* permet de donner l'autorisation à un message d'utiliser les canaux et les tampons. Finalement, le *stockage* définit l'approche utilisée pour mémoriser des messages qui ne peuvent temporairement pas cheminer dans le réseau. Ces divers concepts de réseau seront décrits plus en détail dans le prochain chapitre.

Les NoC transmettent des paquets plutôt que des mots (comme sur les bus). Les fils dédiés aux adresses ne sont plus utiles car l'adresse de destination est maintenant contenue dans le paquet [HEWC04]. Plusieurs transactions peuvent être traitées en parallèle si le réseau fournit plus qu'un canal de transmission entre une source et une destination données.

2.1.3. Interface réseau

La mise en place d'un réseau sur puce doit réduire le temps alloué aux communications. Cependant, le réseau doit être accessible aux blocs qui veulent s'y connecter. Dans un scénario futur pour lequel un concepteur pigera à gauche et à droite dans plusieurs bibliothèques d'IP, cette tâche pourrait se révéler ardue. Les IP peuvent avoir été conçus par des compagnies différentes avec des interfaces différentes et des façons de communiquer tout aussi différentes. Une interface uniforme a l'avantage de libérer le concepteur, en ce sens qu'il n'a pas à faire des suppositions sur le système dans lequel un bloc sera utilisé [WIGO02]. Le but est que tous les composants parlent le même langage lorsqu'ils communiquent entre eux. Dans une architecture de bus, par exemple AMBA, tous les composants sont « enveloppés » d'une couche supplémentaire, cette couche traduisant leurs signaux en d'autres signaux qui seront compris par les autres enveloppes (*wrappers*) des autres composants. Le principe reste le même pour les NoC, où les modules sont enveloppés d'une interface réseau (en anglais, NI pour *Network Interface*). La Figure 2.3 illustre une situation où un processeur, une mémoire, un coprocesseur de même qu'un module matériel. Les principales tâches d'une NI sont :

- cacher aux modules les détails concernant le protocole de communication du réseau en temps que tel. Les modules n'ont pas à se soucier comment l'information est acheminée; ils doivent seulement prendre pour acquis que l'information se rendra à destination.
- convertir le protocole de communication. Pour les modules IP, les communications se font toujours point à point, c'est-à-dire que la source et la

destination se parlent directement. Concrètement, le paquet devra franchir plusieurs routeurs avant de parvenir à destination.

- mettre en paquets les données. Les modules envoient des données brutes auxquelles seront ajoutées des en-têtes et autres informations. Les NI sont donc responsables de l'assemblage, du désassemblage et de la livraison des paquets.

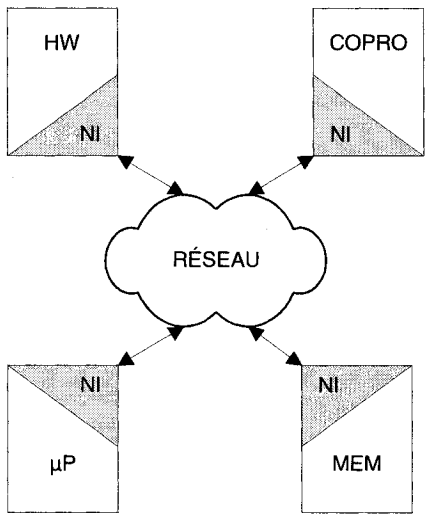


Figure 2.3 : Exemple d'utilisation d'une interface réseau commune pour tous les blocs

2.1.4. VCI

VCI (*Virtual Component Interface*) est un exemple de protocole de communication qui permet de normaliser l'interface d'un maître (initiateur, dans la nomenclature VCI) et d'un esclave (cible). Cette interface a été produite par l'Alliance VSI (en anglais, VSIA pour *Virtual Socket Interface Alliance*), un regroupement d'entreprises et de chercheurs dont la mission est de faciliter la réutilisation de IP [VSIA00]. C'est la spécification de VCI qui a ouvert la voie à la réutilisation des IP dans la conception des systèmes. Depuis, la compagnie *Sonics* a développé son propre protocole de communication, qui est nul autre qu'une extension et une amélioration de VCI et qui est donc très bien perçu par l'Alliance VSI.

2.1.5. OCP

OCP est très proche de VCI, mais fournit en plus une suite d'outils qui permettent de vérifier si les noyaux produits sont conformes à OCP. La communication entre un initiateur et une cible se fait via un paquet. L'initiateur envoie d'abord un paquet requête (*request*) et attend une réponse de la cible (*response*). Normalement, la cible renvoie le même paquet après que les champs appropriés aient été mis à jour (par exemple, la valeur d'une donnée si l'initiateur faisait une lecture). Dans ce cas, c'est le pont OCP qui modifiera le paquet, après avoir reçu les données de la cible via son interface. Par la suite, le paquet OCP devra retourner à l'initiateur en cheminant dans le réseau (Figure 2.4). Le réseau peut adopter différentes topologies: un bus classique, un bus hiérarchique ou un réseau commuté. C'est ce qui permet de découpler (orthogonaliser) totalement le choix des composants du système du choix des interconnexions qui les relieront. Il faut simplement créer le pont OCP – réseau (identifié NI sur la figure) et ce pont pourra être réutilisé pour tous les blocs.

Figure 2.4 : Rôle de l'interface OCP

OCP a été retenu pour ce travail puisqu'un modèle haut niveau de ce protocole était déjà disponible. Les ponts permettant de lier tous les initiateurs et les cibles utilisés dans les simulations à OCP étaient déjà faits, ce qui était un avantage considérable. Par ailleurs, OCP jouit déjà d'une popularité considérable dans le marché, ce qui a joué en sa faveur.

2.2. Concepts réseau

Puisque les réseaux intégrés découlent directement des réseaux à grande échelle, comme les réseaux locaux et Internet, ils auront forcément plusieurs points en commun. Cette section fait un bref rappel de quelques concepts que l'on retrouve dans les réseaux d'ordinateurs.

2.2.1. Réseaux vs NoC

Même si les similarités sont multiples, il n'en demeure pas moins que le concepteur ne peut pas appliquer les techniques de réseautique telles quelles sans tenir compte de plusieurs différences causées par l'application et l'environnement de développement.

On remarque d'entrée de jeu que les NoC se doivent de diminuer au minimum la latence des communications, tandis que les réseaux d'ordinateurs parallèles mettent plutôt l'accent sur la bande passant offerte, puisqu'elles mettent en attente les blocs qui les initient.

Il y aussi une énorme différence dans le déterminisme des communications. Sur une puce, les tâches qui seront exécutées dans les blocs sont connues très tôt dans le processus de design. Un SoC est toujours conçu pour combler un besoin précis et pour effectuer des opérations restreintes. Il est donc possible de prévoir un tant soit peu quelles seront les communications entre les blocs et d'ajuster le réseau en conséquence. Sur un réseau d'ordinateurs, ceux-ci sont habituellement composés de processeurs à usage général qui pourront traiter une gamme d'applications et qui produiront une tout aussi grande variété de trafics. Par ailleurs, le nombre de composants sur un réseau sur puce est fixe et déterminé pendant la période de design. La topologie utilisée (la façon de connecter les composants entre eux) sera donc aussi fixe. Dans un réseau d'ordinateurs, des nœuds peuvent être ajoutés ou retirés à tout moment, ce qui fait en sorte que la topologie change continuellement [WILI03].

De plus, dans un réseau à grande échelle, les algorithmes de routage requièrent une grande tolérance aux fautes, comme par exemple des bris de liens ou des pannes de routeurs. Ce problème se retrouve très diminué sur les réseaux sur puces qui sont moins susceptibles de « tomber en panne » [WILI00].

Finalement, les SoC font face à des contraintes d'énergie et de dissipation de puissance qui sont moins critiques dans les grands réseaux [BEDE02b]. Il est à noter qu'un concepteur a un temps limité pour mettre en marché le système conçu.

2.2.2. Concepts réseau

Les concepts réseau incluent notamment le type de trafic, la commutation, le routage, le multiplexage de données, le contrôle d'erreur, le contrôle de flot, le contrôle de congestion ainsi que l'allocation des ressources [YOO03]. Les prochaines sous-sections se concentrent sur les trois premiers, puisqu'ils auront un impact plus grand sur le type de NoC utilisé dans cette recherche.

2.2.3. Trafic

À une époque où la consommation d'énergie devient un souci important dans la conception des SoC, le contrôle du trafic dans les réseaux devient essentiel puisqu'un bon contrôle permet de mieux gérer la consommation de puissance des composants qui utilisent le réseau [BEDE02b]. Au rythme où vont les choses, le contrôle global du trafic d'informations deviendra impossible car le système se doit de mémoriser l'état de chacun des composants, ce qui est ardu avec le nombre croissant de connexions. Ces composants devront donc initier leurs transactions de manière autonome, selon leurs besoins [BEDE02].

On retrouve deux types de trafic dans les réseaux : trafic garanti (en anglais, GT pour *guaranteed traffic*) et meilleur effort (en anglais BE pour *best effort*). Le trafic de type

GT garantit une certaine bande passante pour un composant, par exemple 4 Mb/s pour un flux de données vidéo. Le trafic GT est nécessaire pour des communications temps réel, tout comme il est approprié pour une intégration rapide des IP. Cependant, ce type de trafic a pour conséquence une faible utilisation des ressources.

Le trafic de type BE donne l'accès aux ressources à la transaction la plus prioritaire. Les architectures conventionnelles de bus basées sur les requêtes et les priorités fonctionnent de cette façon. Le trafic BE ne peut généralement pas être jumelé à une application temps réel car la communication ne peut être garantie, ce qui est une qualité de service essentielle dans ce type d'applications. Par contre, les ressources de communication ont un meilleur taux d'utilisation [YOO03].

Fondamentalement, partager une ressource (BE) et réserver une ressource (GT) sont conflictuels et combiner ces deux types de trafic est une tâche difficile [WIGO02].

2.2.4. Commutation par paquets et commutation de circuit

On retrouve deux façons d'acheminer les données dans les réseaux à grande échelle : la commutation par paquets et la commutation de circuits. La commutation de circuits (Figure 2.5 a) repose sur l'établissement d'une connexion entre deux points. Lorsque la connexion est établie, le module initiateur a un usage exclusif de cette connexion et ce pour toute sa durée. Cela revient à réserver un chemin entre deux blocs. Toute autre transaction nécessitant une portion de ce chemin doit être retardée jusqu'à la libération du lien.

Il existe deux sortes de commutation par paquets. La première, appelée *circuit virtuel* (Figure 2.5b)), fragmente le message en paquets et fixe un chemin entre une source et une destination. Tous les paquets passeront par ce même chemin. Toutefois, une portion de ce chemin peut être partagé avec un autre trafic (paire source-destination différente). On aura donc un multiplexage de paquets sur un même lien. La deuxième, appelée *datagram*,

(Figure 2.5c)) consiste également à séparer l'information en plusieurs fragments élémentaires (paquets) qui voyageront dans le réseau indépendamment les uns des autres. Chaque paquet peut alors prendre un chemin différent de son prédécesseur.

Dans une commutation par paquets, les paquets d'une connexion se retrouvent en compétition avec d'autres pour accéder aux lignes de transmission. La taille du paquet doit par ailleurs être choisie judicieusement. Si la taille est trop petite, le message devra être fragmenté en un nombre trop grand de paquets et le temps de réassemblage sera trop coûteux. D'un autre côté, si la taille du paquet est trop grande, le paquet pourrait bloquer la liaison pendant plusieurs cycles, ce qui retarderait les autres trafics [HEWC04].

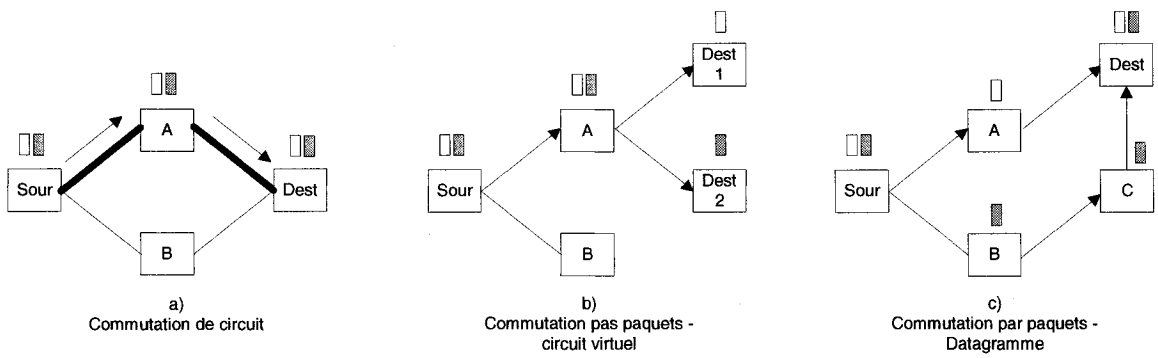


Figure 2.5 : Différentes sortes de commutation

La commutation par paquets peut causer des problèmes reliés à la latence en ce sens qu'un paquet peut parfois prendre quelques centaines, voire un millier de cycles avant d'arriver à destination. Ceci est causé par la distribution statistique des délais associée aux réseaux de commutation par paquets. De plus, puisque les paquets prennent des chemins différents pour se rendre à la même destination, il est fort possible que les paquets arrivent dans un ordre différent duquel ils ont été envoyés. Tel que mentionné ci-haut, le réassemblage des paquets à la destination nécéssite alors un traitement supplémentaire, ce qui entraîne une latence supplémentaire. La commutation de circuits n'a pas ces problèmes puisque la latence dépend seulement de la distance et du temps pris

pour établir la connexion. Les données arriveront toujours en ordre puisqu'elles empruntent le même chemin, peu importe la quantité de transactions concurrentes dans le réseau [WILI03]. Il est à noter que ce type de commutation entraîne un faible taux d'utilisation des liens. Normalement, la commutation de circuits est associée au trafic GT, tandis que la commutation par paquets va de paire avec le trafic BE.

2.2.5. Routage

Le routage détermine le chemin qui sera pris par un paquet de sa source à sa destination. Les facteurs à considérer pour les algorithmes de routage [HEWC04] sont :

- la connectivité : indique s'il est possible de router un paquet pour n'importe quelle paire source-destination.
- adaptabilité : indique s'il existe plusieurs routes pour passer d'une source à une destination.
- anti-interblocage et anti-itinérance : empêche un paquet de bloquer le réseau (*deadlock*) ou d'y circuler indéfiniment sans arriver à destination (*livelock*).
- tolérance aux fautes : indique s'il est possible de router un paquet malgré le bris d'un lien.

Par ailleurs, étant donné les ressources limitées dont une puce dispose, les compromis suivants devront être appliqués aux algorithmes de routage de NoC [BEDE02] :

- déterminisme versus performance moyenne
- robustesse versus agressivité
- complexité et vitesse versus utilisation du canal

Il existe trois algorithmes principaux de routage [YOO03].

2.2.5.1. Stockage et réémission (*store and forward*)

Lorsqu'un routeur a reçu un entièrement paquet, il demande au routeur suivant dans la chaîne s'il est prêt à recevoir le paquet. Si la réponse est positive, le paquet est envoyé; sinon, il est mémorisé. Cette approche est la plus exigeante en termes de mémoire nécessaire et de latence de communication.

2.2.5.2. Par raccourcis (*virtual-cut-through*)

Lorsqu'un routeur a reçu partiellement un paquet, il demande au prochain routeur dans la chaîne s'il est prêt à recevoir complètement le paquet. Si la réponse est positive, l'envoi du paquet est amorcée même si le paquet n'est pas encore complètement reçu; sinon, il est mémorisé. Cette approche nécessite également une quantité significative de mémoire mais permet une plus petite latence de communication.

2.2.5.3. Trou de ver (*wormhole*)

Lorsqu'un routeur reçoit un *flit* du paquet (*flow control digit*, soit la plus petite unité sur laquelle un contrôle de flot peut être fait), il demande au prochain routeur dans la chaîne s'il est prêt à recevoir le *flit*. Si la réponse est positive, le *flit* est envoyé; sinon, il est mémorisé. Cette approche ramène à son minimum la latence de communication et la mémoire requise dans les routeurs. Le désavantage de cet algorithme est qu'il est plus sensible aux interblocages puisque le *flit* de tête peut être bloqué à un certain routeur, ce qui bloquera également tous les routeurs associés aux autres *flits* du même paquet.

2.3. Topologies existantes

Il existe plusieurs façons d'agencer les composants pour former un réseau. Les diverses configurations, jumelées à différents algorithmes de routage, ont des conséquences sur la latence, le filage requis et l'espace occupé par le NoC. Cette section présente d'abord une

brève description des topologies les plus populaires. Par la suite, plusieurs NoC (commerciaux et universitaires) seront passés en revue.

2.3.1. Maille

La maille est probablement la topologie la plus populaire puisqu'elle s'harmonise avec la forme rectangulaire d'une puce. On retrouve les composants (*nodes*) répartis uniformément sur la puce, de même qu'un réseau bidimensionnel de commutateurs (*switches*). Dans sa version la plus simple, chacun des composants est attachée à un commutateur. Toutefois, il est possible d'avoir plus de commutateurs que de composants, tout dépendamment des performances souhaitées et des ressources disponibles. Chaque commutateur est connecté à quatre autres, selon les directions nord, sud, est et ouest. Cette topologie est illustrée à la Figure 2.6 a). Il est également possible d'ajouter un lien entre le premier commutateur d'une rangée et le dernier pour former un tore, ce qui améliore les performances au prix de plusieurs liens supplémentaires (Figure 2.6 b)). Les composants sont identifiés de façon cartésienne avec une coordonnée en X (ligne) et en Y (colonne).

Les mailles sont adéquates pour des communications à grande échelle. Elles sont souvent choisies au détriment des autres topologies en raison de leur coût acceptable en fils, de leur bande passante raisonnablement élevée et dû au fait qu'il est relativement simple de regrouper géographiquement les modules communiquant beaucoup entre eux [WILI03]. Il est donc important de bien exploiter la localité pour profiter des fils courts entre deux commutateurs. On les choisira donc lorsque le délai dans le commutateur approche le délai des fils [CARN00].

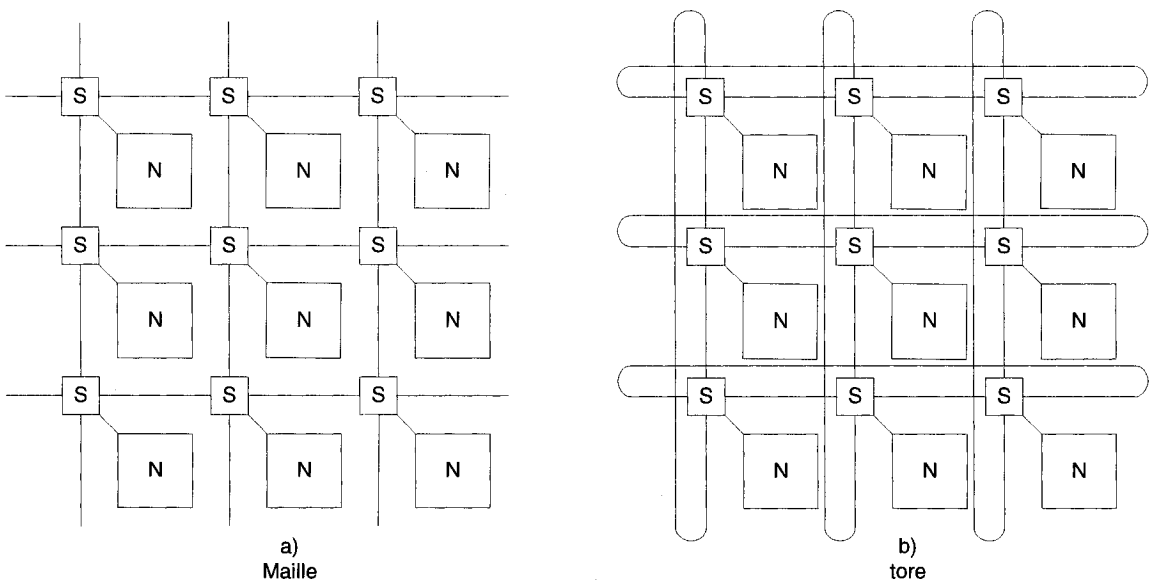


Figure 2.6 : Topologies *maille* et *tore*

2.3.2. Papillon

Les topologies de type papillon (en anglais, *butterfly*) sont des topologies à diamètre logarithmique (leur croissance est proportionnelle au logarithme du nombre de composants qui y sont connectés). Elles nécessitent cependant plus de commutateurs qu'un arbre élargi pour un même nombre de composants (Figure 2.7). Une caractéristique de cet agencement est qu'il n'y a qu'un seul chemin pour n'importe quelle paire source-destination. Ceci peut représenter un risque, qui est minimisé par le fait que les liens sont moins susceptibles de briser dans un réseau sur puce. Les probabilités pour que deux transactions entrent en conflit à un des commutateurs du réseau sont très faibles [CARN00]. Il est toutefois possible de réduire ces probabilités à 0 en utilisant une variante, le *bene*, qui se compose de deux papillons dos à dos.

Les topologies papillon sont moins appropriées que les mailles pour les NoC en raison du coût élevé des fils.

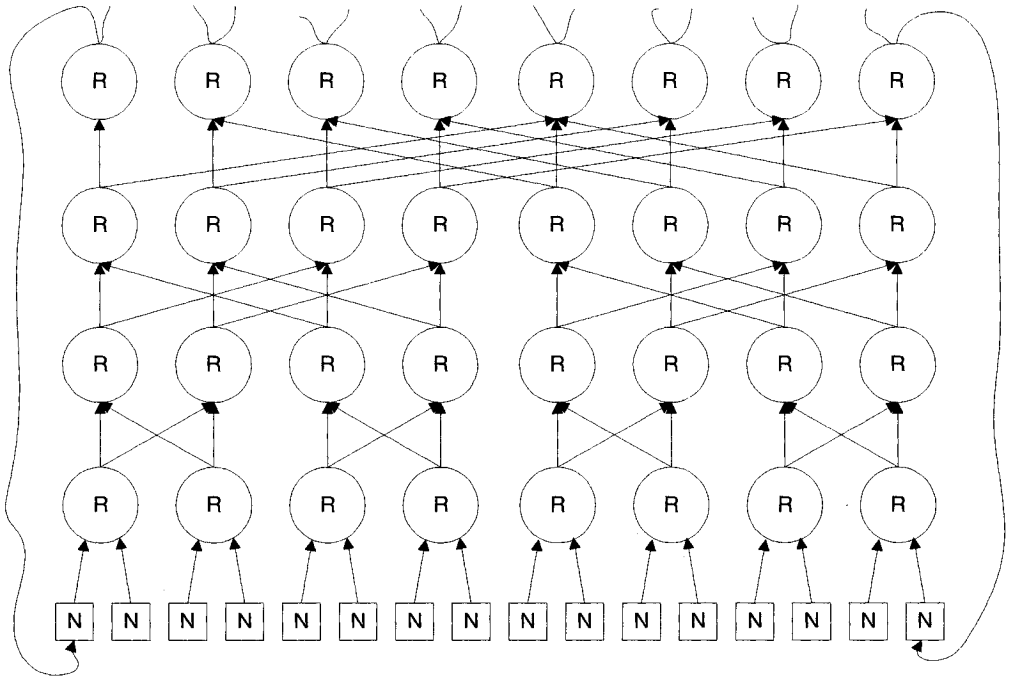


Figure 2.7 : Schéma de la topologie papillon

2.3.3. Arbre élargi

Les arbres élargis, tout comme les topologies dites « papillon », font parti des réseaux à diamètre logarithmique. On peut considérer un arbre élargi comme un papillon auquel on aurait rendu les liens bidirectionnels. Les composants sont attachés à un premier niveau de commutateurs (le standard étant de 4 composants/commutateur). Ces commutateurs peuvent ensuite être rattachés à d'autres commutateurs à un niveau plus élevé, selon le même rapport. C'est le nombre de composants qui dictera le nombre de niveaux de commutateurs à utiliser. Par exemple, dans un réseau à 16 composants (Figure 2.8), deux niveaux de commutateurs sont nécessaires, et un paquet a au maximum trois commutateurs à franchir avant de parvenir à destination. On notera donc une grande amélioration de la latence.

Les réseaux à arbre élargi sont difficilement extensibles, étant donné les longs fils qu'ils nécessitent. De plus, la localité des composants peut être moins bien exploitée. Ils sont tout de même recommandables pour des communications à échelle moyenne, en autant que le délai dans les fils ne devienne pas un problème. Contrairement aux mailles, on choisira les arbres élargis lorsque le délai dans le commutateur est bien supérieur au délai dans les fils.

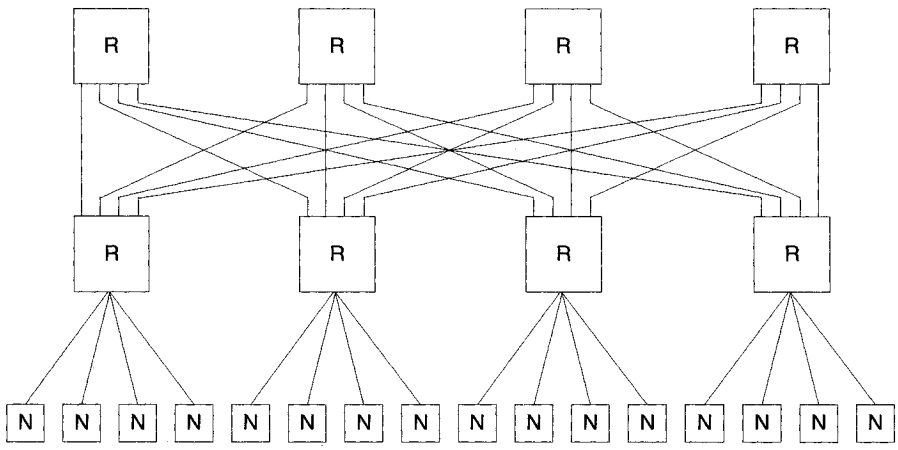


Figure 2.8 : Schéma de la topologie en arbre élargi

2.3.4. Autres topologies

Il existe plusieurs autres topologies qui découlent directement des architectures multi-processeurs. Ces topologies sont plus difficilement intégrables dans les SoC pour diverses raisons. Toutefois, elles sont tout de même présentées dans cette section puisque certains NoC s'inspirent de ces topologies.

Tout d'abord, la topologie *honeycomb* (Figure 2.9 a)) présente un rapport d'un commutateur pour trois composants [JANA01]. Chaque commutateur est connecté à six autres de même qu'à six composants, donc douze connexions. Ceci a pour conséquence une trop grande complexité de commutateur et un routage plus difficile.

Ensuite, la topologie *crossbar* (Figure 2.9 b)) est ce qu'on pourrait appeler l'utopie dans les communications. Une telle architecture permet à un composant d'avoir un lien direct avec tous les autres. Deux blocs qui communiquent voient leurs transactions emprunter des bus « privés », ce qui amène le délai de la transaction à son minimum. Cette topologie n'est pratiquement pas extensible dû à sa complexité astronomique (*N* composants nécessitent *N* * (*N* – 1) / 2 liens). Par contre, elle est idéale pour une communication locale entre quelques modules. Les routeurs sont généralement constitués d'un crossbar.

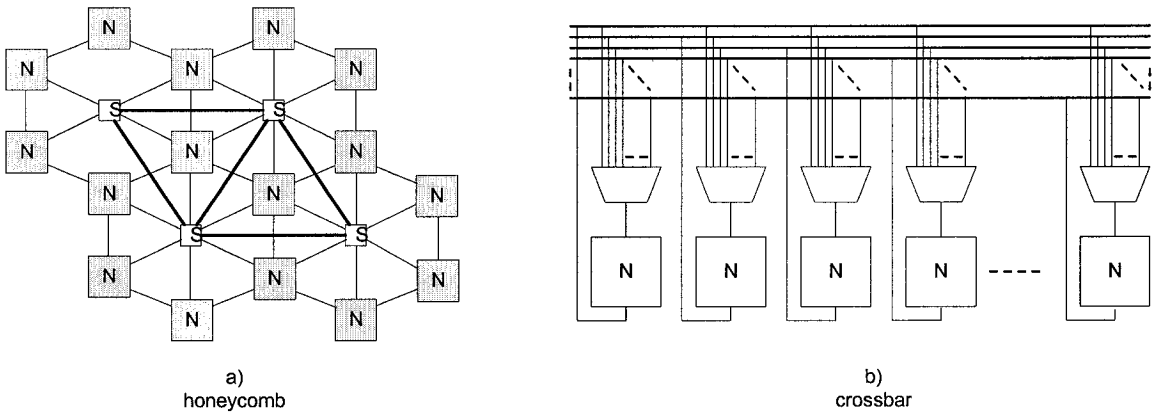


Figure 2.9 : Allure des topologies *honeycomb* et *crossbar*

Le Tableau 2.1 résume les caractéristiques des diverses topologies.

Tableau 2.1 : Caractéristiques des topologies

Topologie	Latence	Complexité	Extensibilité	Échelle de communication
Maille	$O(\sqrt{N})$	O(N)	Oui	Grande
Papillon	O(log N)	O(N log(N))	Difficile	Moyenne
Arbre élargi	O(log N)	O(N log(N))	Difficile	moyenne
Honeycomb	O(N)	O(N)	Difficile	moyenne
Crossbar	O(1)	$O(N^2)$	Non	petite

Les prochaines sections présentent rapidement quelques NoC issus de travaux universitaires ou de sociétés commerciales.

2.3.5. Black-Bus

Blakc-Bus se veut davantage une nouvelle technique pour envoyer des données qu'un NoC proprement dit [AYKJ04]. Les objectifs de Black-Bus sont de retirer les interfaces réseau attachés aux composants de telle sorte que les composants puissent envoyer des données brutes comme sur les architectures conventionnelles de bus et les circuits dédiés. La structure du paquet est remplacée par l'ajout de fils dédiés contenant un identificateur local correspondant à l'initiateur de la transaction. Lorsqu'un nœud débute un transfert, l'identificateur est généré et circule avec les données brutes. Ensuite, quand les données parviennent au prochain routeur, elles sont redirigées d'après les informations contenues dans une table de routage à laquelle on accède via l'identificateur. Une fois les données arrivées au nœud destinataire, l'identificateur permet de déchiffrer le nœud source dans un séquenceur, ce qui permettra au nœud de destination de compléter le transfert de données. Cette technique est indépendante de la topologie du réseau.

2.3.6. ClearConnect®

ClearConnect® est une architecture proposée par la société ClearSpeed™ [CLEA01]. Il s'agit d'un réseau à commutation par paquets conçu spécialement pour une implémentation efficace sur silicium. Le bus traditionnel est remplacé par une chaîne de commutateurs (Figure 2.10), ce qui permet plusieurs transferts simultanés sur différents segments de la chaîne [CLEA03]. De plus, la bande passante augmente avec le nombre de nœuds qui sont connectés à cette chaîne. ClearSpeed™ annonce une bande passante de 12,5 Gb/s pour un bus de 128 bits à 400 MHz. Finalement, la puissance dissipée est proportionnelle à la quantité de données transférées et à la distance qu'elles parcourent. Si aucun transfert n'a lieu, aucune puissance n'est dissipée. Le désavantage de cette topologie est que la chaîne est unidimensionnelle, ce qui rend la latence proportionnelle au nombre de composants.

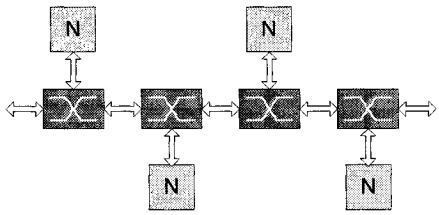


Figure 2.10 : Allure du réseau ClearConnect®

2.3.7. STBus

Le STBus, développé par STMicroelectronics [STMI00], est une architecture modulaire où plusieurs *nœuds* peuvent être instanciés plusieurs fois selon différentes configurations pour former une structure d'interconnexions hiérarchique. Dans la nomenclature ST, un nœud est une matrice de commutation, c'est-à-dire une combinaison de blocs logiciels et de blocs matériels qui servent à acheminer des données d'un pour d'entrée à un port de sortie [BOZZ04]. Dans le STBus, un nœud peut être un bus partagé, un crossbar complet ou un crossbar partiel. Comme mentionné précédemment, le bus est peu coûteux en fils mais lent, le crossbar complet est très performant mais très coûteux, alors que le crossbar partiel se veut un compromis entre les deux extrêmes. Plusieurs nœuds formeront le réseau d'interconnexions global. Le STBus a la particularité d'utiliser trois protocoles, un peu comme AMBA et CoreConnect : les protocoles *peripheral*, *basic* et *advanced.* STBus nécessite donc des convertisseurs de type pour permettre la communication entre deux IP qui n'utiliseraient pas le même protocole. On retrouve aussi des convertisseurs de taille qui permettent la communication entre deux IP qui n'auraient pas la même taille de bus ainsi que des tampons qui agissent comme des étages de resynchronisation entre deux IP utilisant des protocoles différents. Le lecteur voulant en connaître davantage peut se référer aux spécifications [STMI03].

2.3.8. SPIN

SPIN (*Scalable Programmable Interconnection Network*) a été développé en 2000 par le Laboratoire Informatique Paris-6 (LIP6). Il s'agit d'un arbre élargi (voir Figure 2.8), plus

précisément d'un arbre quaternaire. Ce NoC emploie la commutation de paquets par datagramme et un routage adaptatif de type trou de ver (section 2.2.5.3). Ce type de commutation a le désavantage de conduire au désordre dans l'arrivée des paquets, ce qui nécessite un travail supplémentaire pour reconstituer le message. Lorsque la quantité de trafic augmente, le routage trou de ver fait en sorte que plusieurs routeurs peuvent être bloqués simultanément lorsqu'un conflit survient, ce qui fait qu'il est impossible de garantir une latence donnée. SPIN sature d'ailleurs lorsque sa charge atteint autour de 30%. Le projet a été abandonné. Notons que SPIN se veut un NoC de haute performance, mais il est très coûteux en fils étant donné sa structure en arbre.

2.3.9. Hot Potato

Hot Potato, conçu en Suède, a une structure en maille de deux dimensions (voir Figure 2.6a). L'appellation vient du routage *patate chaude*, qui pose comme contrainte qu'un paquet doit obligatoirement changer de commutateur à chaque cycle, quitte à revenir sur ses pas. Ce type de routage a l'avantage de produire des commutateurs de très faible complexité puisqu'il n'est pas nécessaire de mémoriser les paquets. Par contre, lorsque le trafic augmente, le taux de contention augmente considérablement et il est donc possible qu'un paquet chemine vers sa destination sans jamais y parvenir. Une priorité basée sur le nombre de cycles passés dans le réseau peut toutefois corriger ce problème. Malgré l'exploitation de la localité des composants, il est impossible là aussi de garantir une latence maximale pour ce réseau. De plus amples informations sont disponibles dans [NILS02].

2.3.10. SoCIn

Une université du Brésil a elle aussi développé son NoC : SoCIN (*SoC Interconnection Network*). Il s'agit également d'un réseau en mailles ou en torus (Figure 2.6b) utilisant un routage trou de ver de type XY [ZESU03]. Le routage XY consiste à router les paquets d'abord sur une ligne, puis sur une colonne. Chaque paquet prendra donc toujours le

même chemin et ce pour toute paire source-destination dans le réseau. Le routage XY est plutôt restrictif en ce sens qu'il limite l'utilisation de la bande passante offerte de par sa nature. Néanmoins, c'est une des approches les plus simples pour éviter les interblocages et permettre aux paquets d'arriver dans le même ordre. Il est à noter que les commutateurs sont paramétrables sur le degré de connectivité (jusqu'à 5), la largeur du canal (8, 16 ou 32 bits) et la profondeur des tampons (1, 2 ou 3 mots).

2.3.11. ECLIPSE

ECLIPSE (*Embeddeb Chip-Level Integrated Parallel Supercomputer*) est un bon exemple d'utilisation de la maille 2D. ECLIPSE se compose d'une chaîne de processeurs jumelés à des mémoires d'instructions dédiées, de modules de mémoire hautement imbriqués ainsi que d'un NoC en maille clairsemée de haute capacité [FORS02]. Le terme « maille clairsemée » (en anglais, *sparse mesh*) signifie simplement qu'on retrouve plus de commutateurs que de ressources connectées, ce qui permet d'augmenter la bande passante. Plutôt que d'avoir un seul commutateur connecté à une ressource, on retrouve plutôt un *supercommutateur* comprenant quatre commutateurs. À la source, le paquet est acheminé au hasard à un des quatre commutateurs. Par la suite, le message se promène via les supercommutateurs selon un routage XY.

2.3.12. NoCGEN

NoCGEN (*Network on Chip Generator*) est un outil proposé par une université australienne visant à générer un réseau en utilisant des composants modulaires pour former des commutateurs configurables au niveau du nombre de ports, de l'algorithme de routage, de la largeur du canal et de la profondeur des tampons. Un graphe de description représentant les interconnexions entre les commutateurs (c'est-à-dire la topologie) est utilisé pour générer une description haut niveau en VHDL [CHPA04]. NoCGEN se veut un outil semblable à celui développé pour Sonics Backplane, mais appliqué aux NoC. Le

protocole utilisé est semblable au AMBA AHB. À ce jour, seule la maille est supportée comme topologie.

2.3.13. Et les autres…

Il existe plusieurs autres propositions de NoC, la plupart étant issues des universités. Mentionnos d'abord Aethereal [RGRM03], qui supporte à la fois un trafic GT et BE Ensuite, il y a Proteo [SAAN03], qui consiste en une petite bibliothèque de composants paramétrables pour former un grand éventail de topologies, protocoles et configurations. Vient ensuite XPipe [BEBE04], qui veut pousser cette approche à la limite en voulant instancier un NoC pour une application spécifique. D'autres NoC non décrits ici, tels que SoCBUS [WILI03], Nostrum [MNTK04], QNoC [ROVF05] et autres vont dans la même direction et tentent de répondre à un besoin criant de performance à coûts minimum.

CHAPITRE 3 : Le *Rotator on Chip*

Le présent chapitre présente un modèle de réseau intégré sur puce qui a été développé en étroite collaboration avec la société STMicroelectronics à Ottawa. Il est important d'insister sur la notion de *modèle*, puisque ce NoC n'a pas été réalisé physiquement mais qu'il est simplement programmé en C++. Le but fixé au départ est de monter à un niveau d'abstraction plus élevé dans le but de bien évaluer la fonctionnalité d'un NoC de même que de mieux cerner les paramètres qui interviennent dans sa conception. Ainsi, ce type de modélisation permet de recueillir très peu de mesures concernant la puissance dissipée par les composants, la complexité matérielle de ce NoC sur la puce et la fréquence possible d'opération. Par ailleurs, le chemin des données (*datapath*) se retrouve très simplifié puisque l'on fait abstraction des composants de bas niveau comme les multiplexeurs et les registres à décalage. Par conséquent, la taille des paquets, des tampons et autres modules composant le réseau sont simplement approximatifs et leur détermination finale est laissée à de futurs concepteurs qui poursuivront les travaux à plus bas niveau.

3.1. Vue d'ensemble

Le RoC (*Rotator on Chip*) est inspiré du *Token Ring*, un réseau où tous les ordinateurs sont connectés schématiquement en cercle (Figure 3.1). Un jeton (agencement spécial de bits) chemine dans ce cercle. Pour envoyer un message, un ordinateur « attrape » le jeton quand il passe, y attache son message et laisse le jeton poursuivre sa route dans le réseau. Tout comme les architectures simples de bus, un seul message peut circuler à la fois dans le réseau, puisqu'on n'y retrouve qu'un seul jeton.

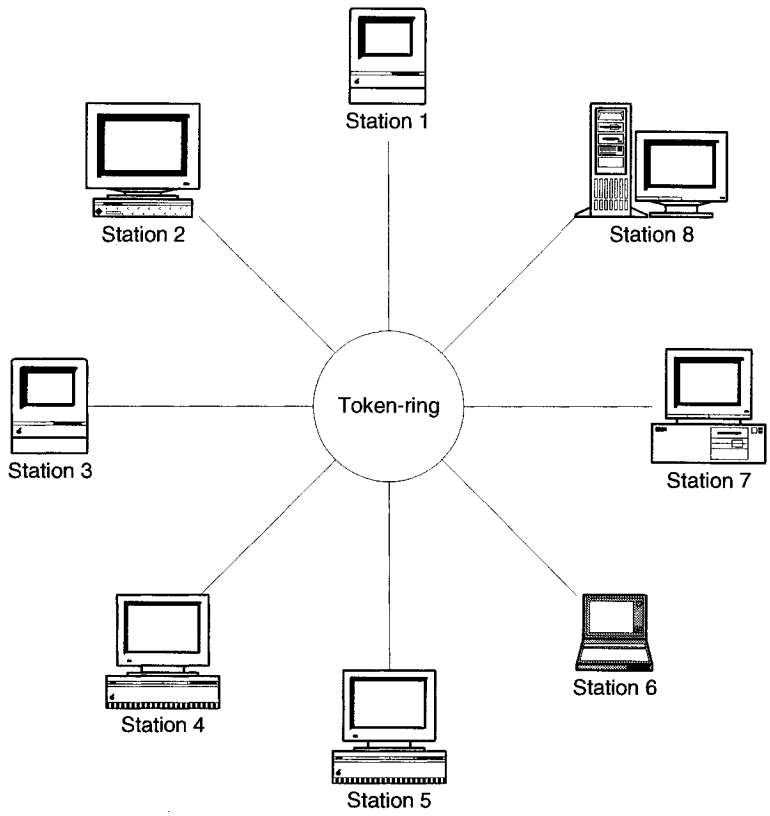


Figure 3.1 : Architecture Token Ring

Le RoC se veut à la base une architecture *Token Ring*, mais à plusieurs jetons, en ce sens que tous les composants qui y sont connectés peuvent envoyer simultanément des messages. Certains pourront faire la comparaison avec un *Token Ring* pipeliné, ce qui n'est pas erroné.

Soit *N* le nombre de ressources à être connectées au réseau. Une ressource, rappelons-le, peut être un processeur, un coprocesseur, une mémoire, un circuit matériel dédié ou un autre périphérique. Chaque ressource est attachée à un *nœud*, qui agit comme interface réseau (voir la section 2.1.3). Cette interface est responsable de transformer les requêtes/réponses des ressources en paquets à acheminer dans le réseau. Les paquets circulent sur *N banques* qui se connectent successivement à tous les nœuds. La vue

d'ensemble du NoC est présentée à la Figure 3.2. Les prochaines sections décrivent en détails les modules du réseau ainsi que leurs rôles.

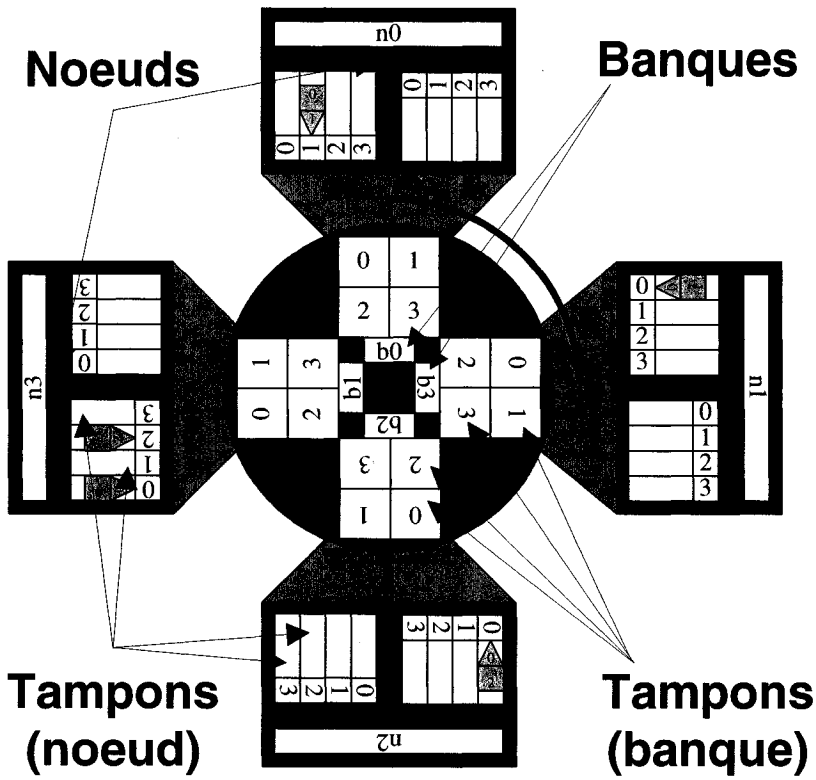


Figure 3.2 : Vue d'ensemble du RoC

3.2. Le noeud

À la base, un nœud est composé de tampons d'entrée et de tampons de sortie. C'est par l'intermédiaire de ces tampons que les paquets sont injectés dans le réseau ou retirés de celui-ci. Comme nous le verrons plus loin (chapitre 4), un nombre plus restreint de tampons pourrait être utilisé avec pour conséquence une légère diminution de performance. La Figure 3.3 décrit la structure d'un nœud. En a), on peut observer les ports d'entrée/sortie du bloc. Un pont OCP est nécessaire puisque la communication entre les nœuds et les ressources qui y sont attachées est analogue à celle qui est représentée à la Figure 2.4.

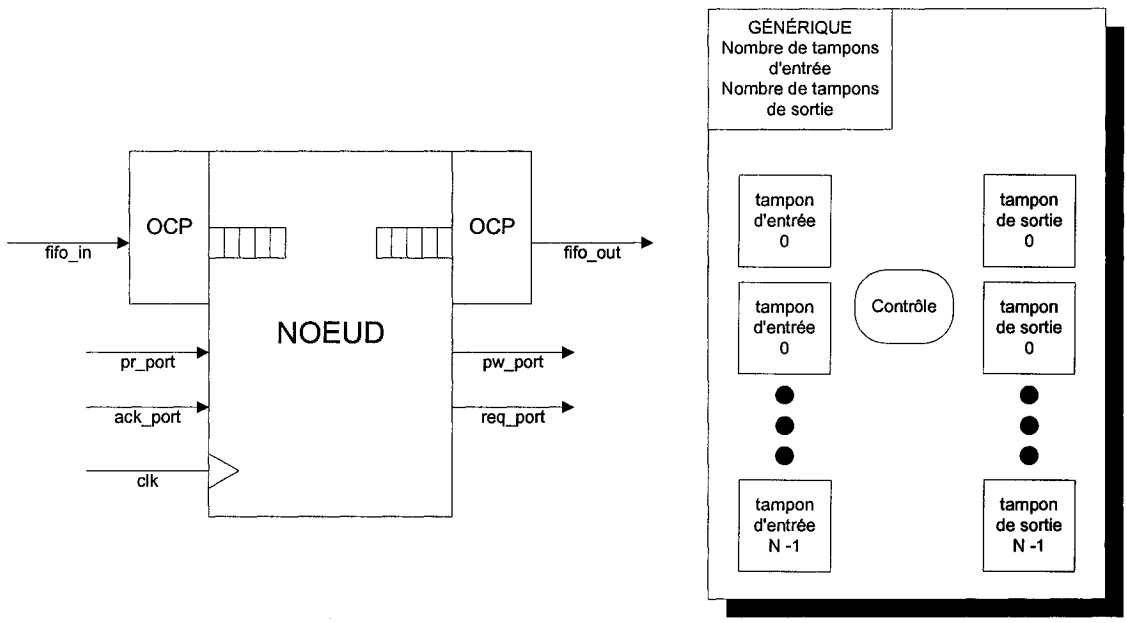


Figure 3.3 : Description du noeud

C'est donc par ce pont OCP que les paquets arrivent et repartent de la ressource. Le port *pr_port* est un port d'entrée qui reçoit un paquet d'une banque alors que le port *pw_port* envoie un paquet à une banque. Les requêtes de transaction générées par le nœud sont acheminées par le port *req_port* alors que les réponses sont reçues par le port *ack_port*. Le Tableau 3.1 résume le rôle de chacun des ports.

Tableau 3.1 : Description des ports d'un nœud

Port	Type	Description
pr_port	entrée	Lecture d'un paquet en provenance d'une banque
pw_port	sortie	Écriture d'un paquet vers une banque
req_port	sortie	Écriture d'une requête vers une banque
ack_port	entrée	Lecture d'un accusé de réception en provenance d'une banque
fifo_in	entrée	Réception d'un paquet en provenance d'une ressource
fifo_out	sortie	Envoi d'un paquet vers une ressource

La Figure 3.3 b) montre ce qui constitue le cœur de l'interface : des tampons d'entrée, des tampons de sortie ainsi que des blocs de contrôle pour effectuer les transactions correctement. Le nombre de tampons est paramétrable et est normalement égal au nombre de ressources connectées au RoC. Ce nombre peut être fixé par le concepteur juste avant la synthèse.

Les rôles du nœud sont donc de :

- Transformer les paquets de type *OCP* provenant des ressources en paquets de type *RoC*
- Permettre aux ressources d'envoyer des paquets de façon transparente
- Construire des requêtes selon l'état de ses tampons (pleins ou vides)

Des détails supplémentaires sur le protocole et sur le cheminement des données sont présentés plus loin dans ce chapitre.

3.3. La banque

La banque joue le rôle de commutateur dans le réseau. La banque contient plusieurs tampons par lesquels les paquets voyagent de la source vers la destination. Une banque contient normalement un nombre de tampons équivalent au nombre de ressources connectées au RoC, comme c'est le cas pour le nœud. Ainsi, chaque tampon est identifié, de 0 à $N-1$.

La Figure 3.4 a) présente le diagramme bloc d'une banque, qui comporte quatre ports principaux, outre celui relié à l'horloge. Ces ports sont complémentaires à ceux d'un nœud. Le port *pr_port* est un port d'entrée qui reçoit un paquet d'un nœud alors que le port *pw_port* envoie un paquet à un noeud. Les requêtes de transaction générées par le nœud arrivent à la banque par le port *req_port* alors que les réponses sont retournées via le port *ack_port*. Le tableau 3.2 résume le rôle de chacun des ports.

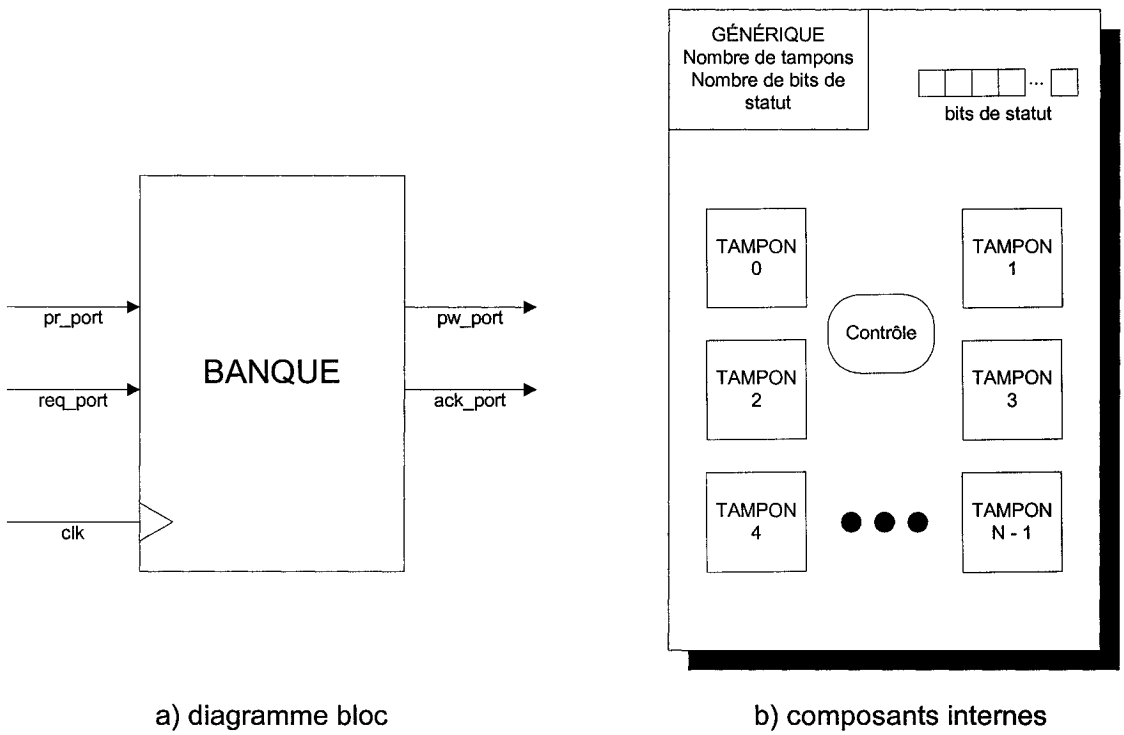


Figure 3.4 : Description d'une banque

Tableau 3.2 : Description des ports d'une banque

Port	Type	Description
pr_port	entrée	Lecture d'un paquet en provenance d'un nœud
pw_port	sortie	Écriture d'un paquet vers un nœud
req_port	entrée	Écriture d'un accusé de réception vers un nœud
ack_port	sortie	Lecture d'une requête en provenance d'un nœud

La Figure 3.4 b) montre le contenu d'une banque. D'une part, des tampons servent à stocker les paquets. D'autre part, la banque incorpore également des bits de statut qui correspondent à l'état des tampons de la banque suivante dans le réseau (banque $i + 1$ pour une banque i). L'état des tampons peut être *vide* ou *plein* (0 ou 1). L'utilité de maintenir ce statut est expliquée ultérieurement. Le nombre de tampons ainsi que le nombre de bits de statut sont deux paramètres génériques. Finalement, un bloc de contrôle gère la cohérence globale du module.

On peut résumer les rôles de la banque ainsi :

- Transporter un paquet du nœud source au nœud destination
- Gérer l'arbitration dans les requêtes (donner la permission ou non au nœud d'envoyer un paquet)
- Maintenir l'état des tampons de la prochaine banque à jour

3.4. Cheminement des données

Soit un ensemble composé de quatre ressources connectées via un RoC. Examinons le cas typique où un processeur veut écrire une donnée quelque part sur une mémoire externe. Soit *N0* le numéro du nœud qui rattache la mémoire au réseau et *N3* le numéro du nœud qui rattache le processeur au réseau. Le processeur met à sa sortie les valeurs d'adresse qui correspondent à l'emplacement de la mémoire où l'écriture se fera ainsi que la donnée à écrire. Ces signaux sont interceptés par le pont OCP (voir Figure 2.4) qui crée un paquet OCP. La Figure 3.5 b) présente tous les champs qui composent le paquet en question. Le rôle de chacun des champs est défini au Tableau 3.3. Ce paquet OCP est ensuite intercepté par le nœud qui en fait un paquet RoC (Figure 3.5 a)). Les champs *source* et *destination* du paquet RoC sont décodés à partir des champs *mConnID* et *mAddr* du paquet OCP. Dans le cas présent, le champ source aura comme valeur 3, tandis que le champ destination aura la valeur 0. La taille d'un paquet RoC peut être estimée à 100 bits, selon le nombre de bits qu'on associe aux différents champs. Cette approximation est cohérente mais n'a pas d'influence sur la suite des opérations puisque le traitement se fait à un plus haut niveau d'abstraction. Elle peut néanmoins fournir une bonne approximation au lecteur intéressé par ce paramètre.

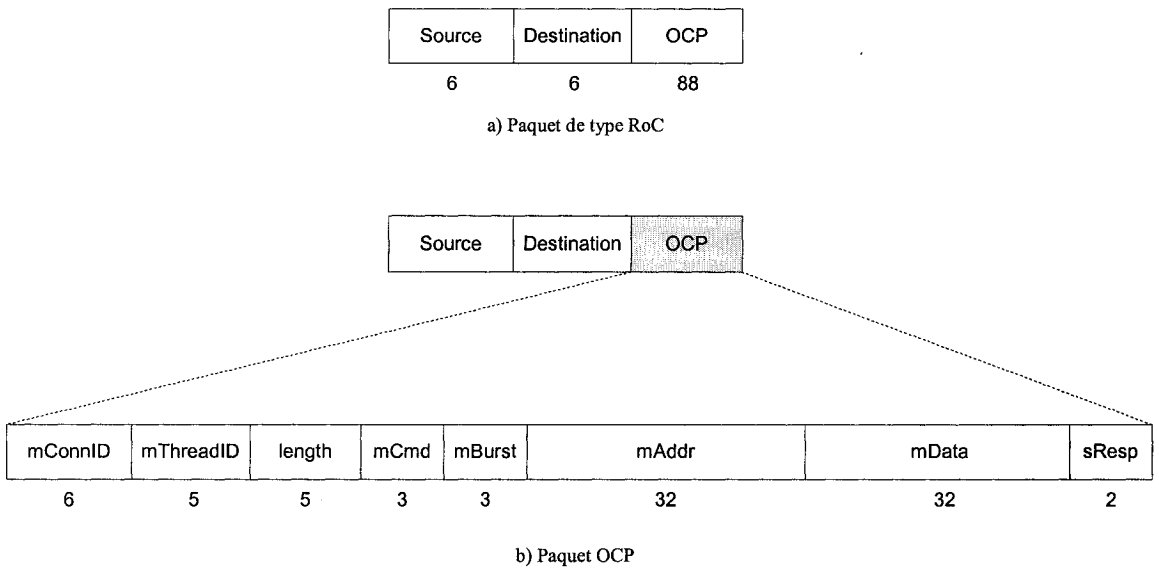


Figure 3.5 : Représentation des paquets

Tableau 3.3 : Description des champs d'un paquet OCP

Champ	Description
mConnID	Numéro d'identification du maître
mThreadID	Numéro de thread
length	Longueur des données à transférer (en mots de 32 bits)
mCmd	Commande du maître à l'interface (typiquement *lecture* ou *écriture*)
mBurst	Code de burst
mAddr	Champ d'adresse
mData	Champ de donnée
sResp	Code de réponse de l'esclave

Lorsque le nœud 3 détecte un paquet dans sa FIFO d'entrée, il va le placer dans le tampon de sortie correspondant à l'adresse de destination (ici, le tampon 0). Au cycle suivant, le nœud envoie une requête à la banque pour lui signifier qu'il a un paquet pour le nœud 0. La banque vérifie l'état des tampons de la banque qui la suit dans la rotation en examinant son bit de statut 0. Si ce bit est à 0, le tampon est libre et un accusé de réception (*ACK*) est envoyé au nœud. Le nœud peut alors envoyer le paquet qui est tamporisé dans le tampon approprié, toujours selon l'adresse de destination (Figure 3.6).

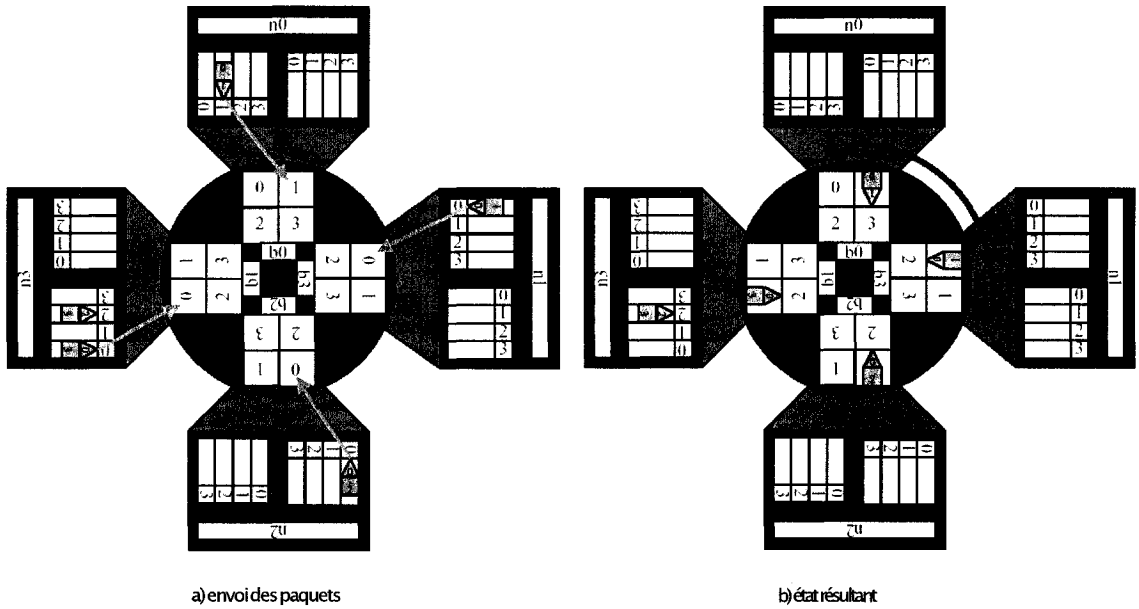
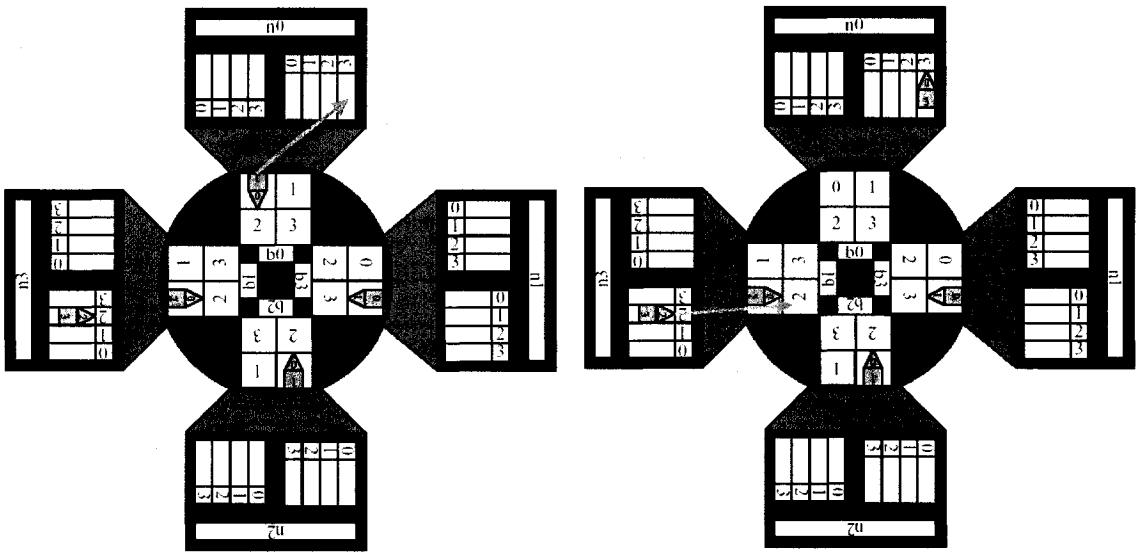


Figure 3.6 : Transfert d'un paquet d'un nœud à une banque

Les paquets « tournent » ensuite dans le sens des aiguilles d'une montre et le noyau de banques mettra *N* étapes à effectuer une rotation complète. À chaque étape, une banque est connectée à exactement un nœud, toujours différent. Le paquet arrive à destination lorsque le numéro du tampon sur lequel il voyage correspond au numéro du nœud qui est connecté avec la banque à une étape donnée. La banque envoie donc le paquet vers le nœud, ce qui vide le tampon. Lorsque le nœud reçoit ce paquet, il va le placer dans le tampon d'entrée correspondant à l'adresse de source du paquet (ici, le tampon 3). Ceci est illustré à la Figure 3.7 a) où le paquet arrivant à destination (noued 0) est transféré dans le tampon 3, puisqu'il a été émis à partir du noeud 3. Il est à noter qu'un nœud peut envoyer un paquet vers une banque au même moment que celle-ci lui en envoie un, comme le montre la Figure 3.7 b).



a) envoi des paquets b) état résultant

Figure 3.7 : Transfert d'un paquet d'une banque à un nœud

L'envoi d'un paquet est conditionnel à l'obtention d'un signal ACK. Toutefois, le nœud n'envoie pas immédiatement le paquet après la réception de ce signal. Cet accusé est plutôt interprété par le nœud de la manière suivante : « **Après la prochaine rotation**, je pourrai envoyer le paquet ayant comme destination 0 à la banque ». Ainsi, un paquet envoyé par un nœud est toujours précédé d'un signal ACK reçu à l'étape précédente. Le nœud procède de cette façon, car cela permet d'envoyer un paquet à la banque tout en envoyant une requête pour le transfert suivant (i.e. à la banque qui suivra selon le sens de la rotation). Cette façon de procéder permet de sauver un cycle sur le traditionnel protocole *handshaking* entre deux ressources. En effet, tandis que ce protocole se compose de quatre étapes (requête, réponse, envoi, confirmation), le protocole RoC comporte trois étapes illustrées à la Figure 3.8. Dans cette dernière, les *wait()* représentent des cycles d'horloge (dans une éventuelle synthèse). Au point 1, le paquet et la requête ont été envoyés par le nœud pendant la première phase et sont disponibles pour la banque. Pendant la deuxième phase, la banque analyse la requête et émet un signal ACK et ce en même temps qu'elle envoie au nœud un paquet qui serait arrivé à destination. Au point 2, ces signaux sont disponibles pour le nœud qui va les recevoir

pendant la troisième phase. Au point 3, les paquets ont cheminé dans le réseau de banques et une nouvelle paire nœud/banque se connecte pour entamer les mêmes processus.

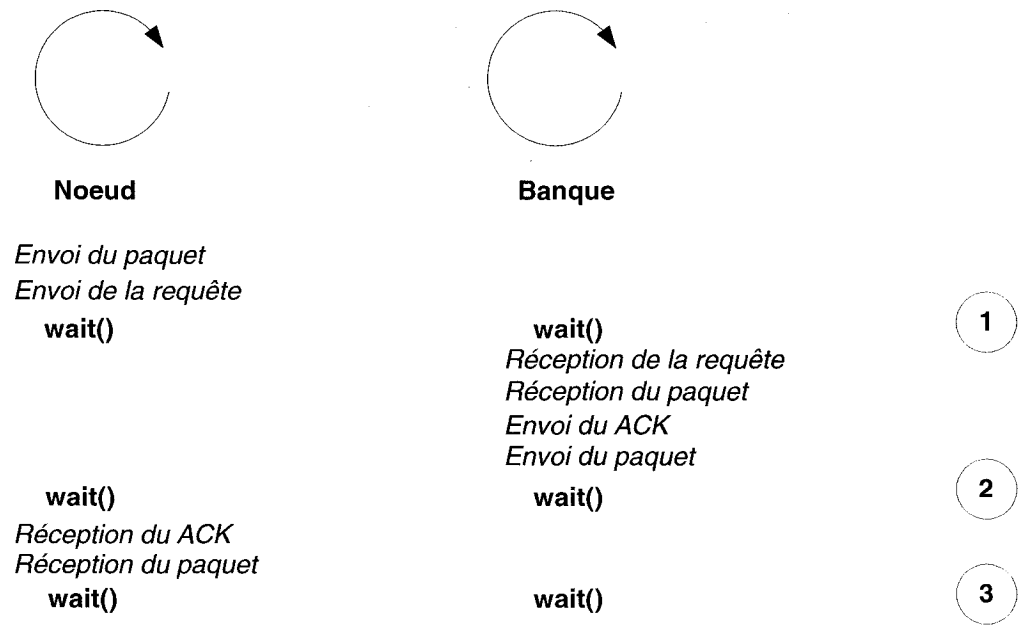


Figure 3.8 : Séquence d'exécution du nœud et de la banque

3.5. Détails d'implantation

Le RoC a été développé en SystemC et a été intégré sur la plate-forme StepNP. SystemC [OSCI01] est une bibliothèque C++ orientée-objet qui permet de modéliser une application à plusieurs niveaux d'abstraction différents. Le système est spécifié à l'aide d'objets tels que modules, processus, canaux et ports. Les modules communiquent entre eux par des canaux via leurs ports. Pour la synchronisation, les processus (ou threads) s'exécutant sur les modules peuvent se mettre en attente d'événements, qu'ils soient synchrones (front montant d'horloge) ou asynchrones (signal d'entrée). L'utilité de SystemC est de modéliser les blocs matériels et logiciels dans le même langage pour obtenir le système entier dans la même spécification. Par la suite, les blocs matériels peuvent être raffinés vers du SystemC synthétisable ou vers des langages de description

de matériel tels que VHDL ou Verilog. Les objets qui ont été utilisés pour le RoC sont les modules (SC_MODULE), les interfaces (SC_INTERFACE), les ports (SC_PORT) et les processus reliés à l'horloge (SC_CTHREAD). Une interface définit les opérations qui sont effectuées via un port. Le canal de communication implante ces opérations. La Figure 3.9 montre les interactions entre les différents objets alors que la Figure 3.10 donne un exemple d'utilisation.

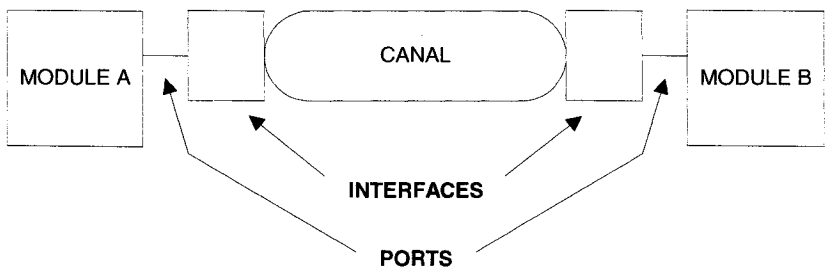


Figure 3.9 : Interaction des modules, canaux, ports et interfaces dans SystemC

```
class A : public sc_module
{
...
sc_port<module_if> read_port;
...
}
```

- déclaration du port
- assignation à une interface

```
class module_if: public sc_interface
{
...
virtual int ReadValue() = 0
...
}
```

- définition de l'opération dans l'interface

```
class canal: public sc_module, public module_if
{
...
int ReadValue()   { return value; }
...
}
```

- implantation de l'opération

```
main()
{
...
module->read_port(canal);
...
}
```

- rattachement du port au canal

Figure 3.10 : Fonctionnement des communications sous SystemC

Le lecteur peut en apprendre davantage sur SystemC en consultant le manuel de référence [OSCI03].

StepNP [PAPB02] est un environnement de développement qui prône la réutilisation de blocs de propriété intellectuelle dans sa méthodologie. Cette plate-forme se concentre davantage sur le développement de processeurs réseaux. Le choix de cette plate-forme a été motivé par plusieurs facteurs. D'une part, elle renferme une bibliothèque de blocs IP tels que processeurs et mémoires. D'autre part, les architectures en développement sont facilement modifiables par l'ajout ou le retrait de composants. De plus, la plupart des outils essentiels au développement d'architectures haut niveau (simulateur, débogueur, analyseur de performance) sont disponibles et simples d'utilisation. Finalement, le code source de cette plate-forme est gratuit et ouvert en raison d'une collaboration entre l'école Polytechnique de Montréal et STMicroelectronics, ce qui est un net avantage dans le contexte académique où les travaux ont été effectués.

StepNP utilise également le protocole OCP, modélisé sous le nom de SOCP (*SystemC OCP*). Considérons à nouveau notre exemple où un processeur (le maître) veut écrire une donnée quelque part sur une mémoire externe (l'esclave). Deux interfaces sont utilisées pour la communication : *SOCPMaster*, pour le maître, et *SOCPSlave*, pour l'esclave. L'interface SOCPMaster définit une seule opération, *putReq()*, qui représente la requête générée par le maître. L'interface SOCPSlave définit quant à elle l'opération *putRsp()*, qui correspond à la réponse de l'esclave. Par ailleurs, le maître découle d'une classe de base appelée SocpMasterBase. Ainsi, lorsqu'un maître désire effectuer une écriture, la fonction *wr* (write) de SocpMasterBase s'exécute. Cette fonction construit le paquet OCP et fait une requête sur le canal par son port (`slavePort -> putReq`). Le maître se met ensuite en attente via un *wait* (écriture bloquante). Le paquet OCP chemine vers la destination selon les règles dictées par le canal. Lorsqu'il parvient à destination, la fonction *putReq* de la mémoire est appelée. C'est à ce moment que le paquet OCP est décomposé et que l'écriture est faite. Par la suite, la mémoire répond au maître en

appelant la fonction *putRsp* du canal pour lui signifier que le travail a été accompli. Le paquet fait ainsi le chemin inverse. Lorsque le paquet revient au point de départ, la fonction *putRsp* du maître est appelée, ce qui le réveillera. L'ensemble de cette procédure est illustré à la Figure 3.11.

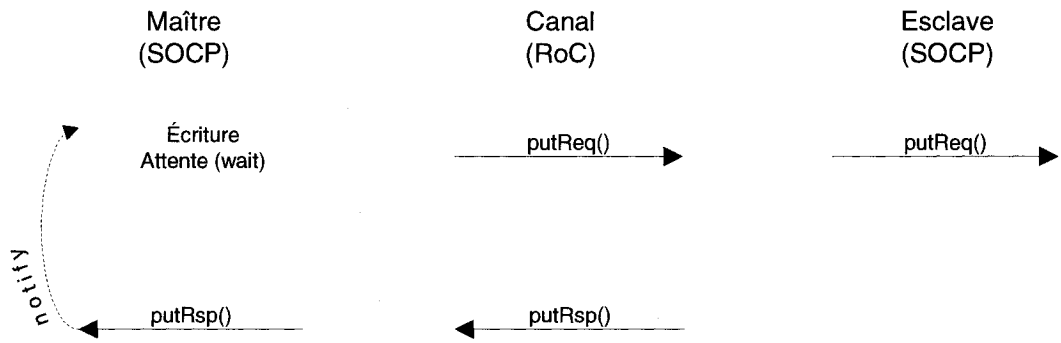


Figure 3.11 : Suite d'appels de fonction menant à une transaction entre maître et esclave

Comme on vient de le mentionner, le contenu des opérations *putReq* et *putRsp* est implanté dans le canal de communication. Ce contenu diffère selon l'architecture utilisée. Par exemple, la fonction *putReq* d'un crossbar appelle directement la fonction *putReq* de l'esclave (acheminement instantané) alors que celle du RoC prend le paquet OCP, lui ajoute les champs source et destination et l'envoie dans le FIFO approprié du bon nœud. Ce n'est que lorsque le paquet RoC arrive au nœud destination que la fonction *putReq* de l'esclave est appelée.

Le diagramme de classes allégé est présenté à la Figure 3.12. La classe *SocpFuncROC* se veut l'intégration du NoC dans StepNP. Le Tableau 3.4 résume les rôles de chacune des classes qui composent le RoC. Il est à noter que les classes *SC_MODULE* et *SC_INTERFACE* proviennent de l'environnement SystemC alors que les classes *SOCPChannelBase*, *SOCPMaster* et *SOCPSlave* sont issues de la plate-forme StepNP.

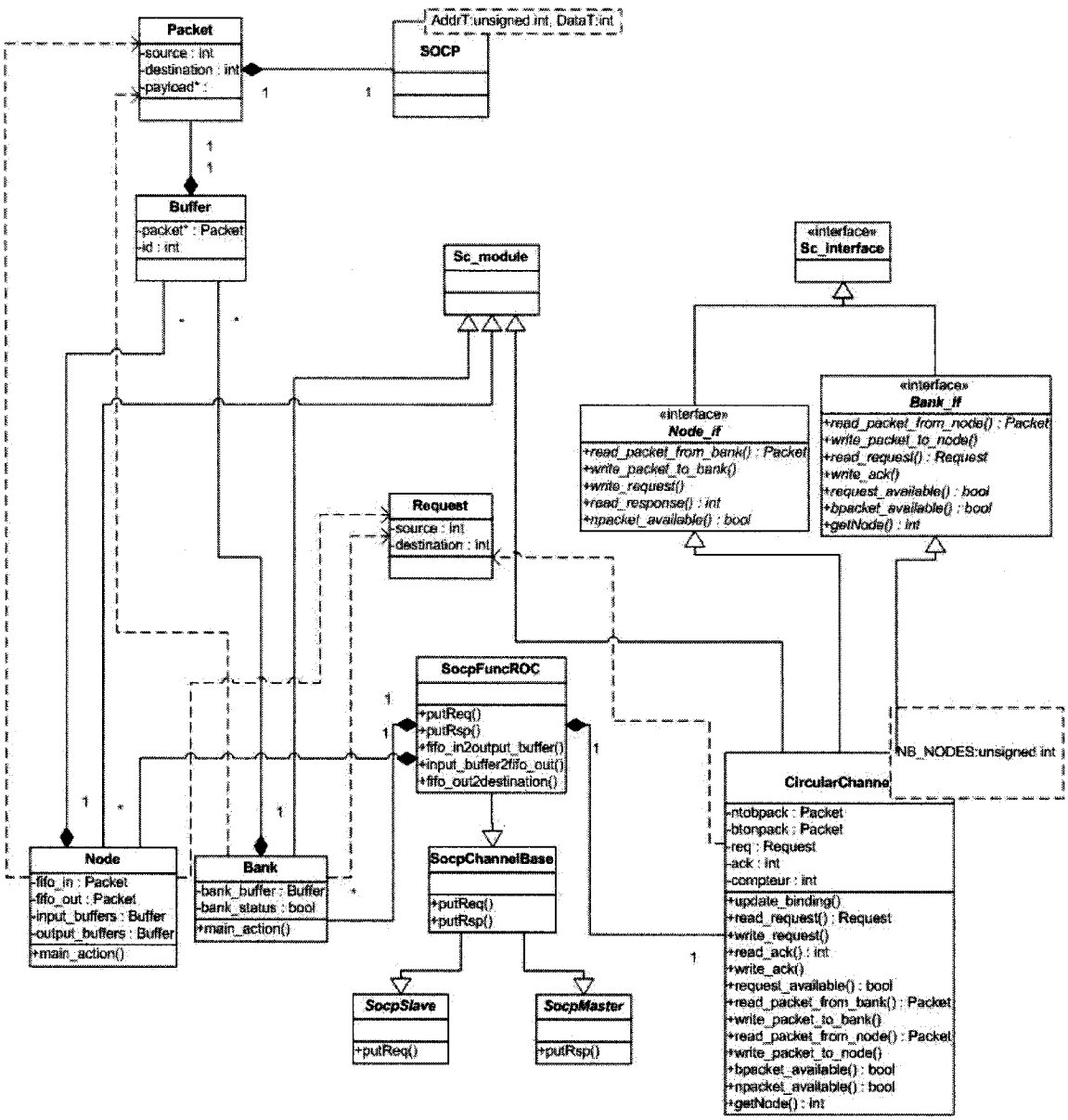


Figure 3.12 : Diagramme de classes simplifié

Tableau 3.4 : Rôle des classes du RoC

Classe	Description
Packet	Définit la structure du paquet RoC
Buffer	Tampon contenant un paquet
Request	Structure d'une requête envoyée du nœud à la banque
Node_if	Définition des fonctions appelées par les ports du nœud
Bank_if	Définition des fonctions appelées par les ports de la banque
Node	Interface réseau qui relie la ressource au RoC
Bank	Objet sur lesquels les paquets circulent dans le réseau
CircularChannel	Le canal de communication proprement dit
SocpFuncROC	Classe qui intègre le RoC dans StepNP (pont RoC – SOCP)

3.6. Caractéristiques du RoC

Le RoC se veut à la base un réseau à commutation par paquets qui utilise un circuit virtuel. En effet, différents paquets cheminant entre une même paire source-destination vont toujours prendre le même chemin. L'ordre d'arrivée des paquets est donc préservé, ce qui sauve le temps pris à reconstituer le message. Cependant, le chemin peut être partagé avec d'autres paires source-destination si la source ou la destination est la même. Par contre, le RoC est intérieurement non bloquant : une connexion entre deux nœuds A et B ne peut pas bloquer une connexion entre deux nœuds C et D et ce pour toute paire A-B différente de C-D.

Les requis en mémoire augmentent rapidement, notamment lorsqu'on ajoute des ressources connectées au réseau. Ainsi, quatre ressources nécessitent quatre banques et donc 4 x 4 = 16 tampons. À huit ressources, ce nombre passe à 64 (croît comme N^2, si N est le nombnre de ressources). La superficie du NoC sur la puce augmente donc de façon quadratique, ce qui peut être un problème. Il est possible d'utiliser un plus petit nombre de tampons sans en affecter les performances. Le chapitre 4 élabore davantage sur cette stratégie et ses implications.

CHAPITRE 4 Optimisation du RoC

Le modèle de base du RoC est fonctionnel et beaucoup plus efficace qu'une architecture de bus. Toutefois, l'algorithme de base et l'utilisation actuelle des composants ne permettent pas d'exploiter pleinement la bande passante que le réseau peut offrir. Quelques améliorations ont été apportées pour augmenter sensiblement les performances sans complexifier le réseau. De plus, il est possible d'utiliser le RoC dans plusieurs configurations qui tentent de répondre à une vaste gamme de trafics et d'applications. Ce chapitre présente les améliorations apportées et décrit ces configurations.

4.1. Requête sous forme d'une matrice de bits (*bitmap*)

Dans sa configuration de base, le nœud vérifie successivement l'état des tampons de sortie selon un algorithme du tourniquet (*round robin*). Lorsqu'il découvre qu'un tampon est plein, le numéro d'identification de ce tampon est mémorisé, de telle sorte qu'au prochain cycle, le nœud reprendra sa vérification où il était rendu. Par exemple, soient huit tampons numérotés de 0 à 7. Soit une première étape où le nœud vérifie les tampons 0, 1 et 2 avant de détecter un tampon plein (le #2). À l'étape suivante (après la rotation), le nœud reprendra sa vérification au tampon 3, puis 4, etc. Le nœud procède de cette façon pour ne pas donner la priorité à une destination en particulier. De plus, c'est une forme d'arbitration simple à implanter et peu coûteuse.

Un cas particulier peut mener à une situation fâcheuse. La Figure 4.1 illustre le cas où un nœud contient des paquets à envoyer pour les destinations 0 et 3. Dans ce cas, l'algorithme du tourniquet mène à la sélection du tampon 3. Une requête est alors construite en conséquence. Lorsque la banque reçoit cette requête, elle examine le statut du tampon 3 de la banque suivante et découvre qu'il est déjà plein. Un refus (NACK) est alors envoyé au nœud. Ainsi, aucun paquet ne sera envoyé par le nœud dans l'étape suivant la rotation. Pourtant, un paquet provenant du tampon 0 aurait pu être envoyé pendant cette étape. Le nœud ne fera donc rien pendant une étape entière, ce qui est

dommageable pour l'efficacité du NoC. Ce cas n'est pas rarissime. En effet, plus le trafic est lourd, plus les probabilités qu'un tampon soit déjà plein sont élevées. Il a donc été impératif de régler ce problème.

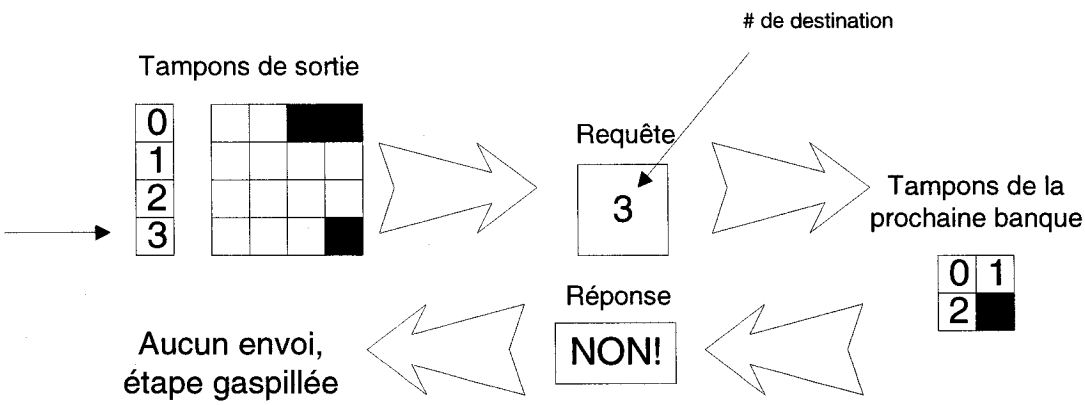


Figure 4.1 : Situation menant à un cycle inutilisé

Une façon plus astucieuse de procéder est de laisser le nœud envoyer une requête qui indique à la banque toutes les destinations à qui un paquet peut être acheminé. Plutôt que de construire une requête en y insérant un numéro de destination, le nœud fabrique une requête qui prend la forme d'un *bitmap*. Ce bitmap contient N bits, un pour chaque destination. Un bit est forcé à 1 si le tampon correspondant est plein et à 0 dans le cas contraire. Lorsque la banque reçoit ce bitmap, elle n'a qu'à le comparer avec le statut des tampons de la prochaine banque, ce qui revient à plusieurs opérations ET BINAIRE effectuées en parallèle. Par la suite, elle choisit une destination qui correspond à la situation *tampon du nœud plein ET tampon de la banque vide* et renvoie le numéro de destination au nœud, plutôt que le traditionnel ACK. Dans le cas où l'envoi demeure impossible, la banque renvoie au nœud l'équivalent de la valeur –1. Ainsi, dans le cas présenté précédemment, le nœud peut envoyer un paquet provenant du tampon 0 plutôt que de perdre son tour, comme le montre la Figure 4.2. Dans sa version actuelle, la banque choisit la destination selon un algorithme du tourniquet. Un autre algorithme pourrait être considéré.

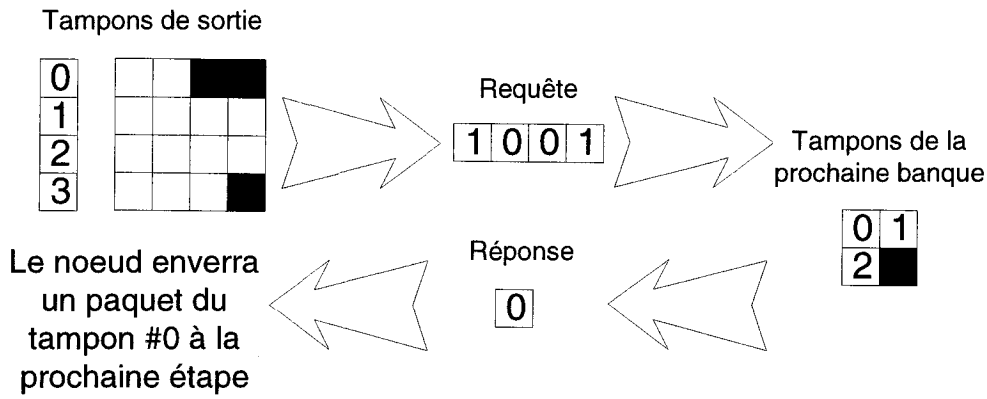


Figure 4.2 : Situation réglée par l'emploi de la requête *bitmap*

Les gains de performance sont notables, puisqu'un paquet est toujours envoyé, dans la mesure du possible. De plus, le coût en matériel est minime, puisque quelques fils supplémentaires sont nécessaires pour supporter les nouvelles requêtes et réponses échangées. Par exemple, à huit nœuds, 8 bits sont requis pour la requête, comparativement à 3, alors que 3 bits sont requis pour la réponse, comparativement à 1. Le chapitre suivant illustre davantage les bénéfices qui découlent de cette astuce.

4.2. Mode rafale

L'utilisation du mode en rafale (*burst*) apporte une performance supplémentaire dans les échanges entre ressources. Par exemple, si les ressources communiquent en utilisant un algorithme de *handshaking*, l'envoi de N paquets nécessite N requêtes, N réponses, N transferts et N confirmations, donc 4N communications. En utilisant le mode en rafale, on abaisse ce nombre à N + 3, soit 1 requête, 1 réponse, N transferts et 1 confirmation. Voilà un avantage indéniable qui rend le support du mode burst primordial.

Le mode burst implique l'envoi successif de plusieurs mots, soit d'un maître à un esclave (écriture) ou d'un esclave à un maître (lecture). L'intégration de ce mode dans le RoC soulève une certaine problématique. En effet, soit l'exemple d'une ressource B voulant envoyer huit paquets en rafale à une ressource A, tel que A est le voisin immédiat à droite

de B. Au même moment, soit un paquet déjà envoyé dans le NoC par une ressource C avec pour destination la ressource A (Figure 4.3 a)). Le premier paquet de B est envoyé sur le NoC puis acheminé vers la ressource A. Il en est de même pour les trois autres paquets suivants. Toutefois, au cinquième envoi, le paquet envoyé par la ressource C atteint le voisin immédiat à gauche de B (Figure 4.3 b)). Si, à la rotation suivante, le nœud B envoie son paquet à A, le paquet envoyé par la ressource C sera écrasé et perdu, ce qui n'est pas souhaitable. Par ailleurs, la ressource B peut décider de ne pas envoyer le paquet pour préserver le paquet déjà en route. Dans ce cas, au niveau de la ressource A, cinq paquets de B arrivent, puis un de C, puis les trois derniers de B. Il faut donc pouvoir garder en mémoire le paquet provenant de C et l'envoyer à la ressource une fois le transfert en rafale terminé. Les tampons de sortie du nœud permettent déjà de mémoriser un paquet. Toutefois, si le trafic est lourd et que le transfert en rafale est sans cesse interrompu, plusieurs tampons sont nécessaires, ce qui est coûteux en espace.

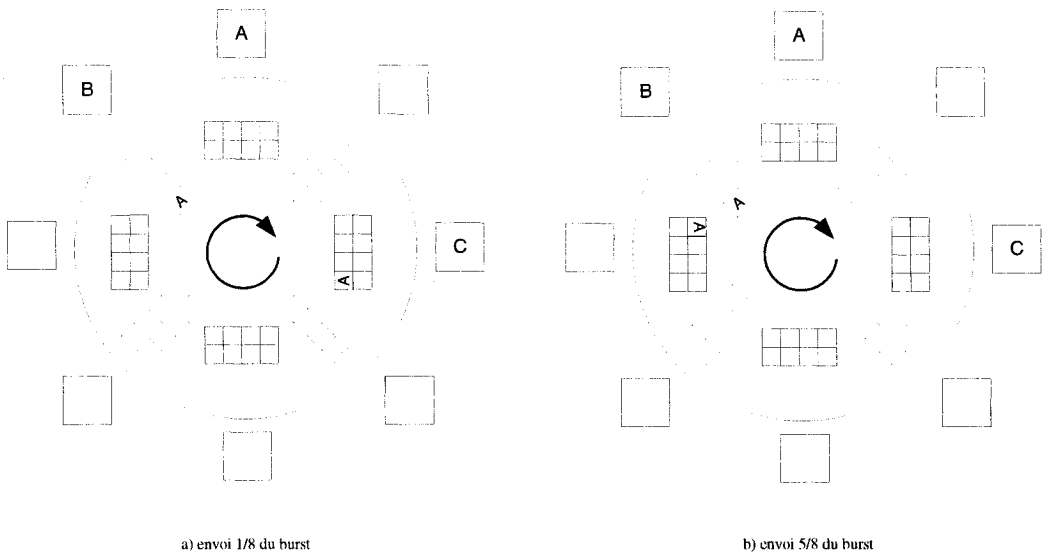


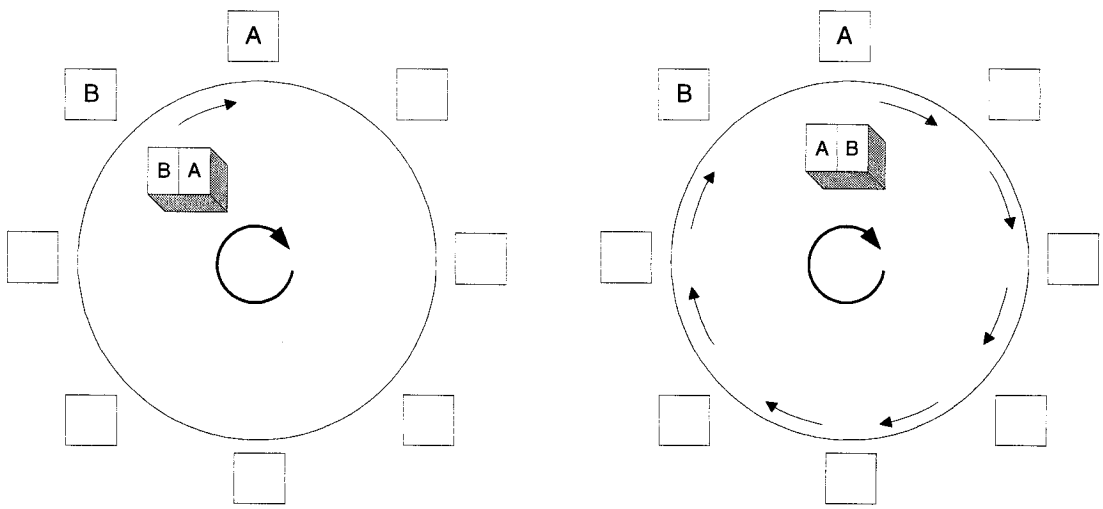
Figure 4.3 : Problème lors d'un envoi en rafale de B à A

Une autre solution à ce problème est de « réserver » tous les tampon A pour la ressource B, un peu comme la commutation de circuit réserve une suite de commutateurs entre deux ressources pour une utilisation exclusive. Toutefois, pour ne pas perdre les paquets déjà présents dans le NoC, il faut attendre une rotation complète du système pour laisser

le temps aux paquets d'arriver à bon port tout en empêchant les autres ressources d'envoyer d'autres paquets de destination A. Dans un RoC à huit ressources, cela implique un délai de sept rotations (ou vingt et un cycles d'horloge selon la figure 2.8). Les répercussions dépendent de la nature de l'application (contraintes de temps, nature temps-réel, etc.). Cette façon de procéder permet néanmoins de garantir une latence d'arrivée des paquets, avec une certaine pénalité prédictible.

4.3. RoC bidirectionnel

Intuitivement, une façon optimale d'agencer une paire de ressources qui communiquent beaucoup entre elles est de les placer côte à côte, de façon à exploiter au maximum la localité des ressources. Cependant, cette configuration pose un problème pour le RoC.



a) envoi de B à A, 1 étape

b) envoi de A à B, 7 étapes

Figure 4.4 : Problème causé par l'architecture unidirectionnelle

En effet, le fait que les paquets tournent dans le sens horaire permet un transfert à latence minimale entre une source B et une destination A (Figure 4.4 a)). Par contre, cela conduit à un transfert à latence maximale lorsque la source et la destination sont inversées, puisque le paquet doit entièrement traverser le réseau avant d'arriver à destination (Figure

4.4 b)). Également, lorsque le trafic augmente, la latence reliée à une transaction entre deux ressources voisines et dans le sens de rotation devient aussi un problème. Soit un RoC à huit nœuds où sept maîtres (processeurs) qui transigent avec le même esclave (mémoire). La mémoire (nœud A) est voisine d'un processeur (nœud B) dans la rotation (voir Figure 4.5). Il est possible d'imaginer une situation où les processeurs écrivent en mémoire plus ou moins simultanément. Lorsque cette situation survient, le processeur B se voit toujours refuser l'accès au tampon A de la banque puisque celui-ci est toujours plein! L'envoi du paquet peut ainsi être passablement retardé puisque les probabilités que le tampon soit plein sont élevées. Cette situation sera illustrée expérimentalement au chapitre 5 (Figure 5.4) où la latence entre le maître 0 et l'esclave 7 (voisins dans la rotation) dégénère lorsque le nombre de threads augmente.

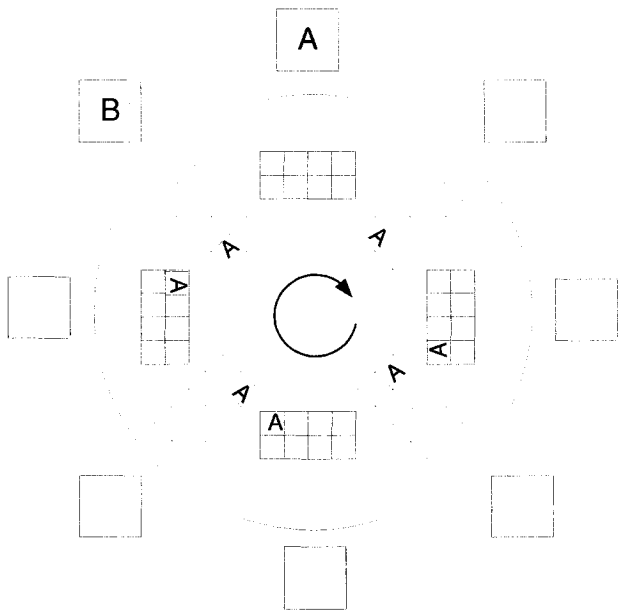


Figure 4.5 : Situation qui conduit à l'incapacité d'un nœud à envoyer un paquet à son voisin immédiat

Une bonne façon de régler ce problème est d'utiliser un rotateur bidirectionnel, tel que le montre la Figure 4.6. Les banques sont regroupées en deux sous-groupes allant dans des directions opposées. Ainsi, chaque nœud est successivement connecté à une banque de

déplaçant dans le sens horaire et à une autre se déplaçant dans le sens anti-horaire. Cette solution permet de profiter de la localité des ressources puisque la latence des transferts entre deux voisins est ramenée à son minimum dans les deux directions. Par ailleurs, si le trafic devient plus imposant, la latence entre deux voisins reste basse puisque la moitié des nœuds ont été connectés à la banque plutôt que la totalité. Dans l'exemple présenté plus haut, on aurait donc trois processeurs plutôt que six qui auraient pu envoyer un paquet sur la banque avant le processeur B. Par conséquent, la probabilité qu'un transfert soit possible est augmentée. Le chapitre traitant des résultats élabore sur les avantages qu'apportent l'utilisation du RoC bidirectionnel. Cette version du RoC implique cependant l'utilisation de canaux full duplex ainsi qu'un contrôleur de banque plus complexe. En effet, ce contrôleur doit tenir compte du sens de rotation de la banque (horaire ou antihoraire) lorsqu'il analyse la requête provenant du nœud. L'arbitre doit négliger la moitié des bits du *bitmap*, c'est-à-dire les bits correspondants aux nœuds les plus éloignés dans la rotation. Ces nœuds seront traités par une banque allant dans le sens opposé.

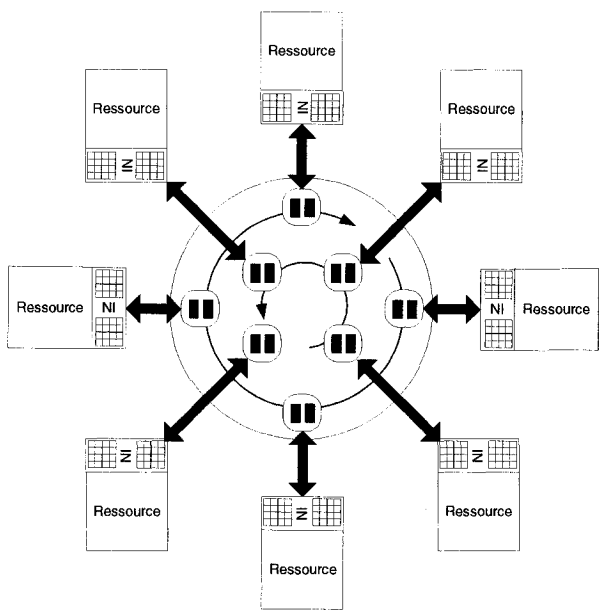


Figure 4.6 : RoC bidirectionnel

4.4. RoC hiérarchique

Lorsque le nombre de nœuds devient relativement élevé, la latence moyenne augmente sensiblement, même avec un RoC bidirectionnel. Ceci est le principal désavantage de la topologie du réseau en anneau par jeton (*Token Ring*). L'objectif est de garder la latence à un niveau respectable, nonobstant le nombre élevé de nœuds qui peuvent être connectés au réseau. Cela suggère l'utilisation d'une structure plus hiérarchique, à l'image des récentes architectures de bus. Le RoC hiérarchique (Figure 4.7) permet d'atteindre cet objectif. Un rotateur dans sa version initiale est appelé satellite et plusieurs satellites sont connectés entre eux via un satellite central. Chaque satellite permet d'incorporer N/S nœuds, où N représente le nombre total de ressources à connecter et S représente le nombre de satellites utilisés. Des interfaces « spéciales » sont utilisées pour acheminer un paquet d'un satellite à un autre. Cette interface peut être attachée à un coprocesseur dédié ou un processeur qui communique très peu avec les autres ressources afin de minimiser le trafic entrant et sortant de ce nœud. Le pire cas découlant de cette topologie survient lorsqu'un paquet nécessite N/S étapes pour atteindre l'interface, S – 1 étapes pour atteindre le bon satellite et N/S étapes pour atteindre le nœud destination. Ainsi, pour traverser ce réseau, un nombre maximal de 2N/S + (S – 1) étapes sont requises, comparativement à N – 1 étapes pour un RoC traditionnel et à N/2 étapes pour un RoC bidirectionnel. Le Tableau 4.1 présente le nombre d'étapes nécessaires selon la configuration utilisée.

Tableau 4.1 : Nombre maximal d'étapes requises pour l'acheminement d'un paquet

Configuration	Latence	8 nœuds	16 nœuds	32 nœuds	64 nœuds
RoC traditionnel	N – 1	7	15	31	63
RoC bidirectionnel	N / 2	4	8	16	32
RoC hiérarchique	2(N/S – 1) + S – 1	7*	9*	17*	21*
RoC hiérarchique bidirectionnel	N/S + S/2	5*	6*	10*	12*

*** Indique qu'il faut ajouter deux (2) au nombre d'étapes calculé pour avoir un résultat plus véridique**

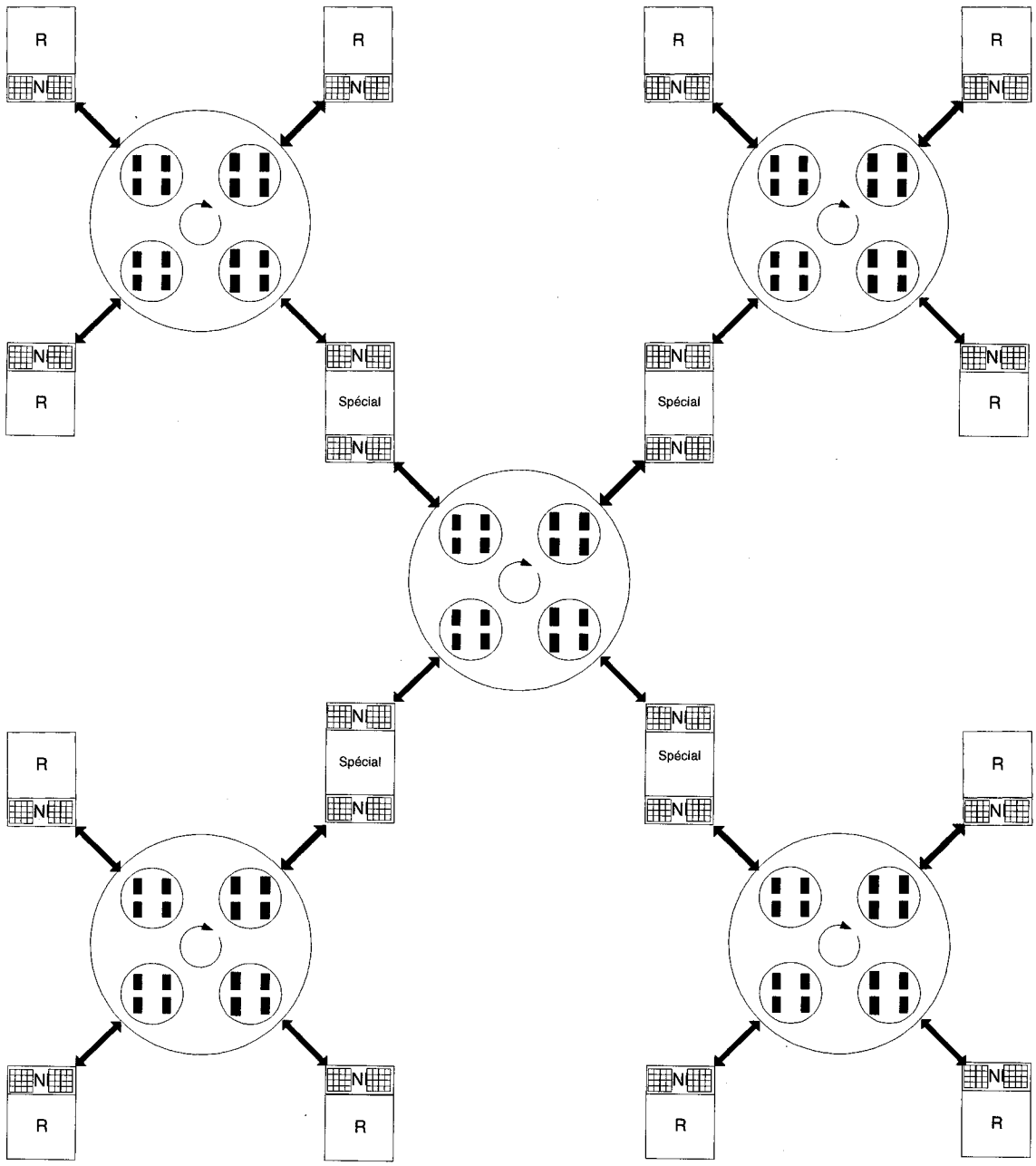


Figure 4.7 : RoC hiérarchique

Un délai supplémentaire de deux étapes doit cependant être considéré lorsqu'un paquet passe par le module central. Ce délai provient du fait aux subtilités d'implémentation qui font que le paquet doit passer du tampon d'entrée au tampon de sortie de l'interface

spéciale pour entrer dans le module central. Le délai pour en sortir est le même puisque le paquet doit changer de tampon une seconde fois lorsqu'il atteint le bon satellite. Tel qu'indiqué au tableau 4.1, l'astérisque (*) indique donc qu'il faut ajouter deux (2) au nombre d'étapes calculé pour avoir un résultat plus véridique.

Le RoC hiérarchique permet d'augmenter la bande passante du réseau en diminuant la latence. De plus, une allocation intelligente des ressources permet de garder la plupart des transactions locales, c'est-à-dire sans passer par le module central. Par ailleurs, le RoC devient moins coûteux en tampons lorsqu'on le hiérarchise. En effet, malgré le fait que S^2 tampons supplémentaires sont nécessaires pour le satellite central, les satellites comptent pour leur part $(N/S)^2$ tampons, plutôt que N^2. Pour un RoC à 128 nœuds, 16384 tampons sont nécessaires dans la version classique. En utilisant 8 satellites de 16 nœuds chacun, ce nombre passe à 2376! Cela démontre que le RoC hiérarchique rend le réseau hautement extensible. Jumelé au RoC bidirectionnel, il permet d'atteindre des performances plus qu'intéressantes. De par sa nature générique, il est possible de configurer les satellites pour supporter un nombre quelconque de nœuds. Par exemple, pour 29 ressources, il est possible d'utiliser 5 satellites (4 de 6 nœuds et 1 de 5 nœuds). Si une topologie en maille était utilisée, il faudrait utiliser 30 commutateurs (5 x 6), ce qui engendrerait un gaspillage d'un commutateur.

Le prochain chapitre présente les résultats complets sur les différentes configurations.

CHAPITRE 5 : Résultats et analyse

La validation du modèle du RoC s'est faite en trois étapes. Tout d'abord, le cheminement d'un seul paquet a été analysé pour bien comprendre les interactions entre composants ainsi que les caprices du simulateur utilisé. Une fois cette étape complétée, le RoC a ensuite été soumis à une batterie de tests fonctionnels qui ont permis de cerner les limites du réseau en fonction de différents paramètres configurables. Pour effectuer ces tests, trois modèles de trafic ont été considérés : « aléatoire », « au même endroit » et « voisin ». Ces types de trafic seront décrits plus tard. Finalement, le RoC a été utilisé pour simuler une application multimédia, soit le MPEG4.

Les autres modèles de RoC (bitmap, bidirectionnel et hiérarchique) ont été soumis aux mêmes tests pour isoler leurs avantages et inconvénients. Afin de situer les performances de ces différents modèles, des simulations supplémentaires ont été effectuées avec les modèles *Token Ring* et *Hot Potato* (maille bidimensionnel décrit à la section 2.3.9).

5.1. Description des simulations fonctionnelles

Des simulations fonctionnelles ont été faites à l'aide du simulateur SystemC dans l'environnement StepNP. Un petit programme, nommé *funcTest*, instancie des maîtres (modèles génériques de processeur) et des esclaves (mémoires). Les processeurs ne font aucun traitement; ils ne font qu'écrire une donnée quelque part sur une mémoire pour la relire immédiatement après, de façon à symboliser sa consommation. Les transactions sont bloquantes, c'est-à-dire que la lecture est faite lorsque l'écriture est complètement terminée, selon le cheminement illustré à la Figure 3.11. Par ailleurs, il est possible d'augmenter le trafic dans le réseau en incrémentant le nombre de threads roulant sur chaque processeur. Ce nombre peut être spécifié à la ligne de commande. Les threads exécutent tous le même code. Les maîtres et les esclaves sont ensuite attachés au NoC

pour que la simulation débute. L'ajout d'un paramètre de compilation permet de changer un NoC pour un autre très aisément.

5.1.1. Détails sur l'environnement

La période de l'horloge est fixée à 3 ns, ce qui correspond à un NoC opérant à 333 MHz. Cette configuration peut sembler optimiste mais il s'agit seulement d'une référence de temps. Ainsi, tous les NoC simulés roulent à la même fréquence.

Pour les simulations, l'interface réseau des différents RoC a été configurée en mode maître/esclave, en ce sens qu'elle dessert un maître et un esclave, comme le montre la Figure 5.1.

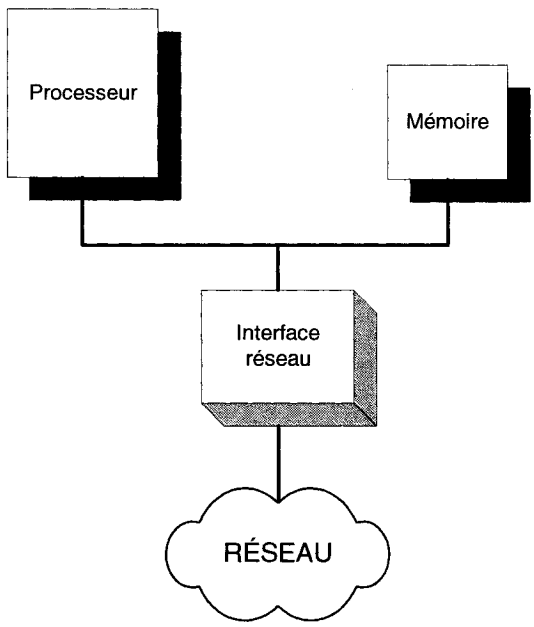


Figure 5.1 : Utilisation d'une interface maître/esclave

Par conséquent, une simulation à huit nœuds implique huit processeurs et huit mémoires, identifiés de 0 à 7.

5.1.2. Types de trafic

Trois types de trafic ont été employés pour les simulations fonctionnelles. Chaque trafic permet de mettre en évidence différentes caractéristiques du modèle de base ainsi que de ses variantes.

5.1.2.1. Trafic aléatoire

Les processeurs écrivent et lisent des données aléatoirement dans une des mémoires. Un processeur n'est pas autorisé à écrire/lire dans la mémoire avec laquelle il partage une interface réseau. Par exemple, dans un système à huit nœuds, le processeur 0 peut écrire dans les mémoires 1 à 7.

5.1.2.2. Trafic « même endroit »

Tous les processeurs, sauf un, écrivent et lisent des données dans une seule et même mémoire. Le processeur qui partage l'interface réseau avec cette mémoire ne fait rien. Par exemple, à huit nœuds, les processeurs 1 à 7 écrivent et lisent dans la mémoire 0 pendant que le processeur 0 est dans un état inactif.

5.1.2.3. Trafic « voisin »

Chacun des processeurs écrit et lit des données dans une mémoire qui lui est voisine dans la rotation. Par exemple, toujours pour un système à huit nœuds, le processeur 5 écrit et lit exclusivement dans la mémoire 4 ou la mémoire 6 alors que le processeur 7 écrit ou lit exclusivement dans la mémoire 6 ou la mémoire 0. Cette configuration représente grossièrement l'exploitation de la localité des ressources.

5.1.3. perNOC

Les mesures de performance ont été prises grâce à un outil d'analyse de performances de NoC : perNOC (pour *performance NOC*). Cet outil [FOAL03] est intégré dans StepNP et permet d'obtenir des résultats d'analyse de performance tels que la latence, la bande passante et la contention. Lorsqu'une transaction est initiée (fonction *putReq*), cette transaction est interceptée par perNOC qui peut ainsi la mémoriser sans son registre. Lorsque la transaction se termine, perNOC en est informé, ce qui lui permet de calculer les différentes valeurs mentionnées précédemment. Le résultat est affiché à l'écran par une ligne de commande, où certaines métriques sont affichées selon la demande de l'usager. Un exemple d'affichage est présenté à la Figure 5.2. L'axe des ordonnées affiche le délai moyen d'une transaction entre un maître et un esclave. Cette figure illustre donc quatre maîtres (étiquette Master # sur l'axe des abscisses) transigeant avec autant d'esclaves (une couleur par esclave selon la légende).

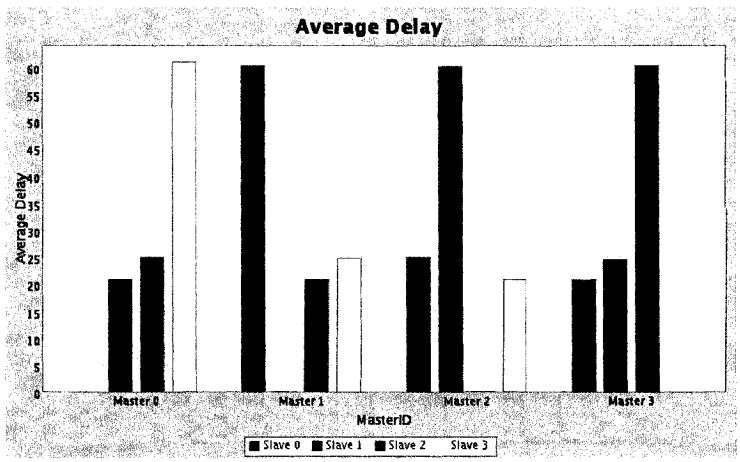


Figure 5.2 : Affichage des résultats de performance avec perNOC

5.2. Résultats des simulations fonctionnelles

Voici en rafale quelques résultats qui permettent de caractériser le RoC et de le situer parmi les autres NoC selon ses forces et ses limitations.

5.2.1. Trafic aléatoire

Le trafic aléatoire a pour but de cerner les performances du NoC. Ce type de trafic est relativement approprié pour les réseaux à grande échelle, mais n'est pas relié à la réalité des SoC, où les transactions sont déterministes, connues et répétitives. Il s'agit néanmoins d'un repère intéressant pour une évaluation initiale.

5.2.1.1. Résultats de base

On peut remarquer la tendance en escalier qui découle de l'architecture *Token Ring* : plus la ressource est située loin dans la rotation, plus le temps de transfert augmente (Figure 5.3). La latence est donc proportionnelle au nombre de ressources connectées dans le réseau. Ceci constitue un désavantage par rapport à la topologie en maille où la latence est proportionnelle à la racine carrée du nombre de ressources et à la topologie en arbre où la latence est proportionnelle au logarithme du nombre de ressources connectées. Le concepteur devra donc utiliser tous les dispositifs intégrés au RoC de façon à garder l'écart minimal entre les autres topologies.

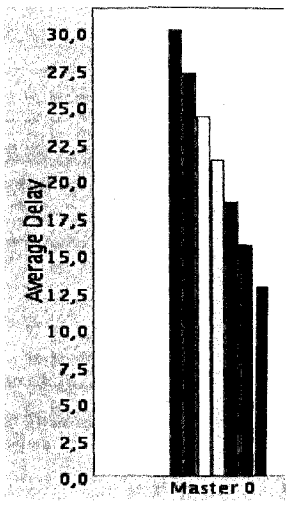


Figure 5.3 : Résultat typique du RoC pour la latence

5.2.1.2. Effets de l'augmentation du nombre de nœuds

L'ajout de nœuds supplémentaires entraîne une augmentation de la latence moyenne des paquets dans le réseau. Cela va de soi puisque les ressources sont géographiquement plus éloignées, en moyenne. Par contre, la bande passante offerte augmente proportionnellement avec le nombre de nœuds. Ceci est caractéristique des NoC, par oppsition aux architectures de bus. Le Tableau 5.1 montre par ailleurs que la bande passante associée à un nœud reste relativement constante. Par conséquent, le nœud n'a pas à se soucier du nombre de ressources connectées dans le réseau lorsqu'une transaction doit être effectuée; il y a toujours une place réservée pour lui sur le réseau.

Tableau 5.1 : Effet de l'ajout de nœuds sur la bande passante du RoC

Nombre de noeuds	Bande passante (Gb/s)	Bande passante normalisée (Gb/s)
4	3,60	0,90
8	6,89	0,861
16	12,72	0,795

5.2.1.3. Effets de l'augmentation du nombre de threads

L'objectif de ce test est de constater jusqu'où le RoC peut garder un comportement normal lorsque le trafic augmente. Le Figure 5.4 montre que pour huit nœuds, la bande passante sature aux environs de 7 Gb/s. Un agencement optimal des composants sur la puce permettrait de profiter autant que possible de cette bande passante.

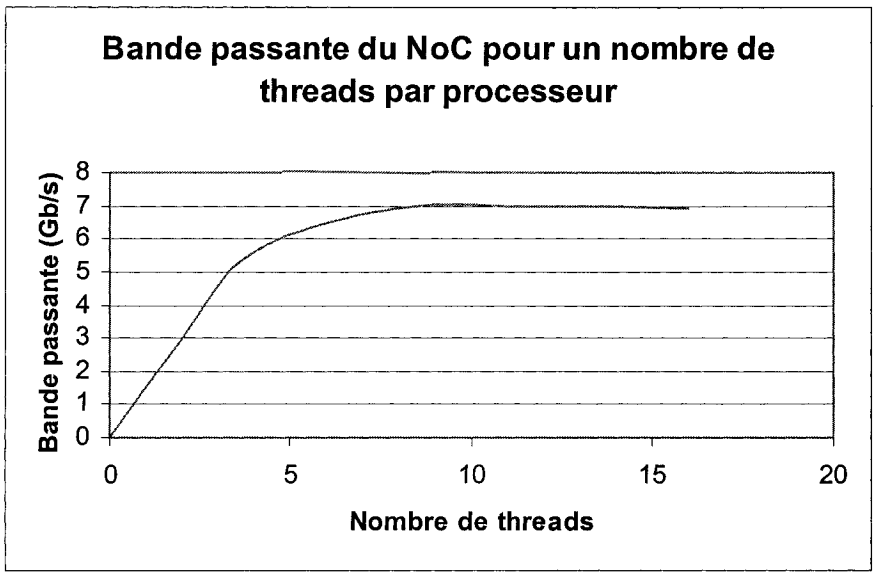


Figure 5.4 : Saturation de la bande passante

Par ailleurs, lorsque le trafic devient imposant sur le réseau, le phénomène de saturation décrit à la section 4.3 se produit effectivement, tel qu'illustré à la Figure 5.5. Pour un certain nombre de threads par maître (1, 2, 4, 8 et 12 sur la figure), on note que la latence dégènere lorsque ce nombre dépasse 8.

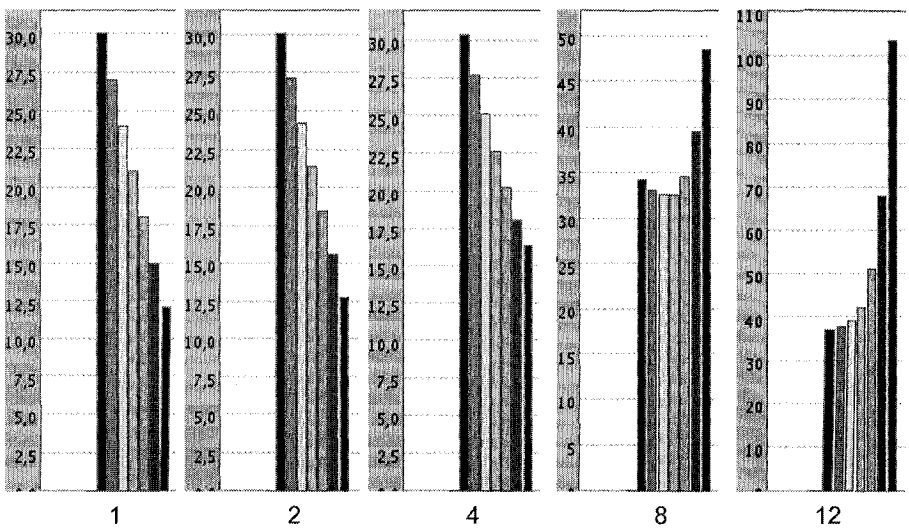


Figure 5.5 : Dégénérescence de la latence pour un transfert entre deux ressources voisines

En effet, tel que discuté à la section 4.3 lors de la présentation du RoC bidirectionnel, la situation observée provient du fait que le tampon rattaché à une ressource voisine dans la rotation est très souvent déjà occupé par un paquet envoyé par une autre ressource. Le paquet demeure alors beaucoup plus longtemps dans le tampon de sortie du nœud avant de pouvoir être envoyé. Le RoC bidirectionnel peut régler en partie ce problème puisqu'une banque dessert un nombre plus restreint de nœuds.

5.2.1.4. Effets de l'utilisation de la requête *bitmap*

Pour un même nombre de cycles d'exécution, plus de transactions sont effectuées lorsqu'une requête de format *bitmap* est employée plutôt qu'une requête standard (pour une destination en particulier). Tel que présenté dans la section 4.1, la requête bitmap permet au nœud d'envoyer un paquet plus souvent puisque la banque considère l'ensemble des tampons plutôt qu'un seul lors de l'analyse de la requête. La Figure 5.5 montre une augmentation significative du nombre de transactions complétées alors que la bande passante du NoC passe de 6,89 à 8,62 Gb/s, soit une augmentation de 25%. De plus, le coût matériel de ce dispositif est minime (se référer à la section 4.1 pour les détails).

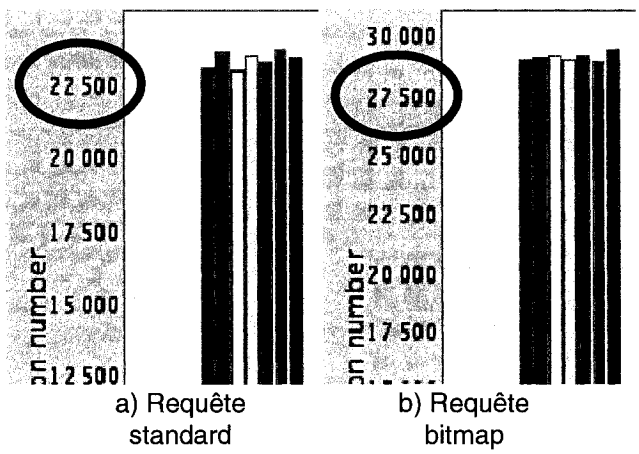


Figure 5.5 : Augmentation de la bande passante effective avec une requête bitmap

5.2.1.5. Utilisation du RoC bidirectionnel

Le RoC bidirectionnel permet de diminuer le temps que prend un paquet à cheminer dans le réseau, tel que prévu. Une petite pénalité s'ajoute toutefois en début de transfert. Par conséquent, le paquet doit attendre une étape supplémentaire une fois sur deux, en moyenne. En effet, pour que le paquet soit envoyé, le nœud doit être connecté à une banque qui se dirige dans la bonne direction. Si ce n'est pas le cas, une seule étape est perdue puisqu'il est certain que le nœud sera connecté à une banque valide (qui va dans le bon sens) à l'étape suivante. Cette pénalité est largement compensée par le fait que le paquet doit traverser au plus la moitié des nœuds pour arriver à destination, plutôt que la totalité dans la version originale du RoC. Une allure en escalier double résulte donc de cette nouvelle configuration, comme l'illustre la Figure 5.6.

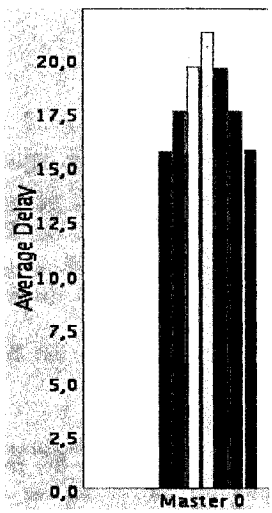


Figure 5.6 : Résultat typique du RoC bidirectionnel pour la latence

5.2.1.6. Utilisation du RoC hiérarchique

Les simulations d'un RoC hiérarchique ont été effectuées avec deux satellites de quatre nœuds chacun. Les résultats montrent que le RoC hiérarchique est plus ou moins performant pour un trafic distribué. Ceci s'explique par le fait que les paquets doivent changer de satellite à deux reprises avant d'arriver à destination. On constate donc deux

étapes où le paquet stagne car le nœud qui y est associé doit faire une requête pour ensuite envoyer le paquet l'étape suivante. Voilà pourquoi la Figure 5.7 montre un écart plus grand entre les trois premières destinations, situées sur le même satellite, et les quatre autres, connectées à l'autre satellite. Ceci explique aussi pourquoi on obtient une latence moyenne supérieure à celle que l'on obtient en utilisant un RoC standard. Par contre, avec un plus grand nombre de ressources, le RoC hiérarchique devient meilleur puisque les paquets arrivent à destination plus rapidement, et ce malgré la pénalité engendrée par l'attente supplémentaire dans les nœuds transitoires.

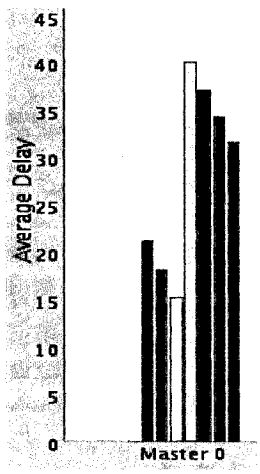


Figure 5.7 : Résultat typique du RoC hiérarchique pour la latence

Par ailleurs, étant donné que tous les paquets externes (dont la destination se trouve sur un autre satellite) doivent transiter par le même nœud, ce nœud peut devenir le goulot d'étranglement dans le réseau lorsque le trafic augmente. En effet, si un paquet ne peut accéder au satellite central en raison d'un tampon plein, un autre paquet arrivant au nœud transitoire doit être stocké dans un FIFO. Voilà pourquoi le RoC hiérarchique est à proscrire pour un trafic très distribué. Cependant, les sections suivantes montrent qu'il est plus approprié pour un trafic local.

5.2.1.7. Effets de la diminution du nombre de tampons sur une banque

Une des préoccupations que soulève le RoC est la surface qu'il peut occuper sur la puce avec le nombre de tampons requis sur chaque banque qui croît de façon quadratique. La consommation de puissance et la taille qui en résultent peuvent être source d'inquiétude pour le concepteur. Des simulations ont été effectuées pour valider le taux d'utilisation moyen des tampons ainsi que l'effet d'une réduction du nombre de tampons sur les performances. Le Tableau 5.2 montre que le taux d'utilisation des tampons tourne autour de 50% avec un RoC utilisant le bitmap. Cette donnée concorde avec la supposition qu'un paquet parcourt la moitié des nœuds pour arriver à destination, en moyenne. Avec l'utilisation du RoC bidirectionnel (incluant la requête bitmap), le taux d'utilisation passe à 25%, étant donné que les paquets parcourent maintenant le quart des nœuds totaux, en moyenne. Ces résultats laissent présager qu'il est possible de réduire considérablement le nombre de tampons et donc l'espace occupé par le NoC sans en dégrader les performances.

Tableau 5.2 : Taux d'occupation moyen des banques pour un RoC à 8 nœuds

NoC	Moyenne d'utilisation de tampons à saturation	Pourcentage
RoC	2,64	33.0%
RoC bitmap	3,52	44.0%
RoC bidirectionnel	1,89	23.6%
RoC hiérarchique	1,35	16.9%

En effet, les simulations montrent que la réduction du nombre de tampons affecte peu la latence des paquets, même dans le cas d'un trafic élevé (huit threads par processeur). Dans le cas du RoC bidirectionnel, ce n'est que lorsque le nombre de tampons occupés atteint 25% du nombre total que le réseau perd de son efficacité, comme le présente la Figure 5.9. Néanmoins, il faut apporter plus de soin au contrôleur de la banque étant donné qu'il est impossible d'associer une destination à un tampon. Les tampons deviennent alors partagés. Une solution simple pour éviter la confusion est de comparer

le numéro du nœud connecté à la banque avec un registre indiquant si un des paquets présents dans les tampons a ce numéro comme destination. Le registre doit également indiquer dans quel tampon le paquet en question se retrouve afin de transférer le bon paquet au nœud destination. Cette façon de procéder est commune dans la plupart des réseaux existants. Par ailleurs, une modification doit être apportée à l'arbitre de la banque puisqu'il ne doit pas y avoir deux paquets de même destination sur une seule banque. Cette situation est impossible dans la version standard du RoC mais elle pourrait survenir dans le cas où les tampons ne sont plus assignés à des destinations spécifiques. Étant donné que le nœud ne peut recevoir qu'un paquet à la fois, un paquet de même destination se trouvant sur une banque devrait effectuer une autre rotation complète avant de pouvoir être délivré, ce qui dégraderait la performance. Pour empêcher ce problème, des *flags* pourraient être utilisés pour indiquer les destinations déjà couvertes par les paquets se trouvant sur une banque à un temps donné.

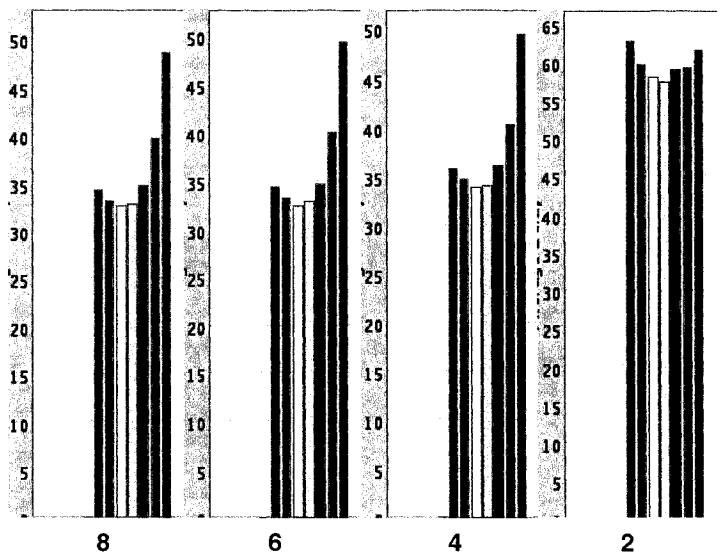


Figure 5.8 : Effet de la diminution du nombre de tampons sur la latence avec un RoC bidirectionnel à 8 noeuds

5.2.1.8. Comparaisons avec les réseaux *Token Ring* et *Hot Potato*

La Figure 5.9 illustre la bande passante offerte par différents réseaux selon le nombre de ressources qui s'y connectent. Tel que prévu, un réseau conventionnel comme le *Token Ring* sature très rapidement et l'ajout de ressources n'a aucun effet sur la bande passante et donc dégénère les performances. Les autres réseaux se comportent de façon extensible, en ce sens que leur bande passante est proportionnelle au nombre de ressources connectées. Pour un trafic aléatoire, le RoC bidirectionnel a le dessus sur le RoC standard, alors que le RoC hiérarchique performe moins bien. Outre la version hiérarchique, les différentes configurations du RoC se comparent avantageusement au réseau en mailles Hot Potato (section 2.3.9).

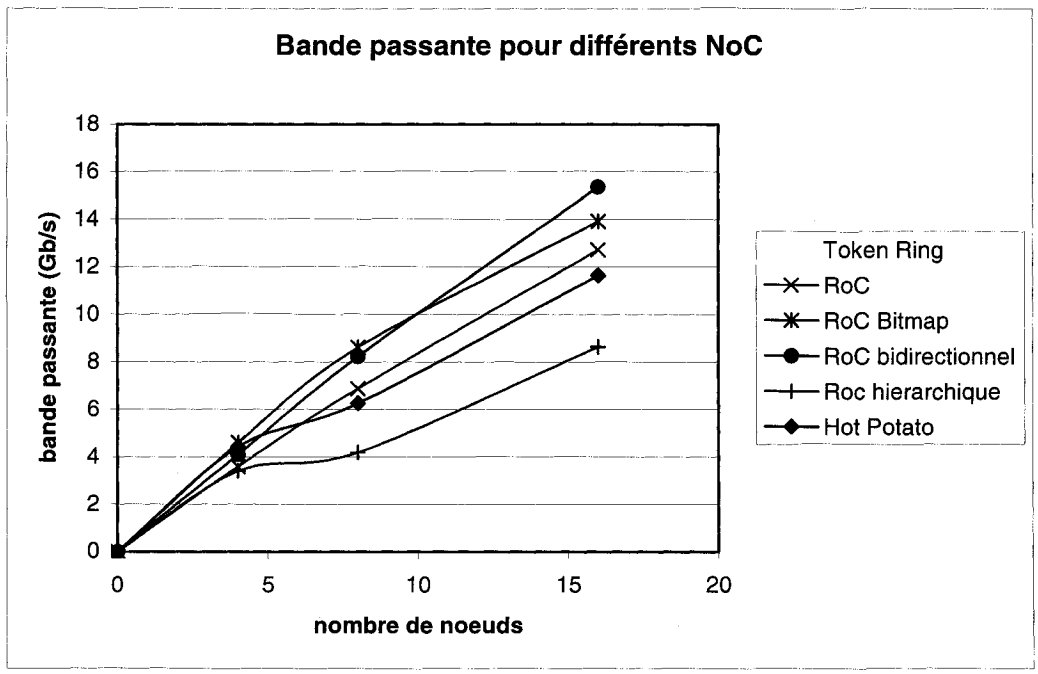


Figure 5.9 : Bande passante pour différents NoC

Il faut par contre faire très attention aux résultats présentés par la Figure 5.9. Tout d'abord, ces résultats découlent de l'hypothèse selon laquelle les commutateurs de tous les réseaux transfèrent un paquet à la même fréquence (333 MHz dans la simulation).

Pour mesurer la réelle bande passante des réseaux sur puce, il faudrait travailler à un niveau d'abstraction plus bas, où les résultats de synthèse permettraient de chiffrer exactement ce que les différents réseaux peuvent donner. Cependant, les résultats obtenus sont très encourageants pour le RoC et ses variantes puisqu'ils se démarquent du réseau en maille même en présence d'un trafic aléatoire. Finalement, chaque commutateur du réseau est constitué de cinq tampons et d'un crossbar 5 x 5. À 16 ressources, 80 tampons sont donc nécessaires, ainsi que 16 crossbars. Un RoC bidirectionnel à 16 ressources nécessite 64 tampons et très peu de logique supplémentaire, ce qui le rend gagnant pour l'espace occupé et la puissante dissipée. Ces estimations sont formulées sous toutes réserves. Tel que mentionné plus haut, des simulations à plus bas niveau devront être effectuées pour les valider. Toutefois, cette tâche dépasse le cadre de la présente recherche et devra être faite dans des travaux futurs.

Par ailleurs, une caractéristique intéressante des NoC est le pourcentage d'utilisation de ses composants. Cette métrique, appelée *load* en anglais, se définit comme le degré de sollicitation des interfaces entre le réseau et les ressources qu'il dessert. Pour le RoC, en supposant une étape de trois cycles d'horloge, le taux d'utilisation est équivalent à :

Nombre de threads * 3 / latence moyenne en nombre de cycles

Pour le Hot Potato, le taux d'utilisation sera égal à :

Nombre de threads / latence moyenne en nombre de cycles

puisque les commutations se font à chaque cycle. La Figure 5.10 montre le taux d'utilisation du réseau pour différents NoC avec un trafic aléatoire. Tout d'abord, comme prévu, le modèle du *Token Ring* sature très rapidement puisqu'une seule requête peut être traitée à la fois. Le RoC dans sa forme classique sature à environ 66%, ce qui se compare avantageusement à un réseau comme Hot Potato, qui sature à 57%, ou à SPIN, qui sature

à 28% [ACGM03]. Par ailleurs, la configuration bidirectionnelle du RoC permet de réduire la latence moyenne d'un transfert et ainsi d'approcher un taux d'utilisation de 83%, ce qui est considérable dans un contexte où l'accent est beaucoup mis sur la réutilisation des composants. Le taux d'utilisation diminue toutefois lorsqu'on utilise le RoC dans sa version hiérarchique. Tel que mentioné plus haut, ce phénomène s'explique par le fait que les paquets doivent souvent changer de satellite, ce qui crée un goulot d'étranglement au niveau des ponts entre satellites. La Figure 5.10 vient donc confirmer une fois de plus que le RoC hiérarchique se comporte très mal en présence d'un trafic aléatoire et qu'il offre de meilleures performances pour des transactions majoritairement locales.

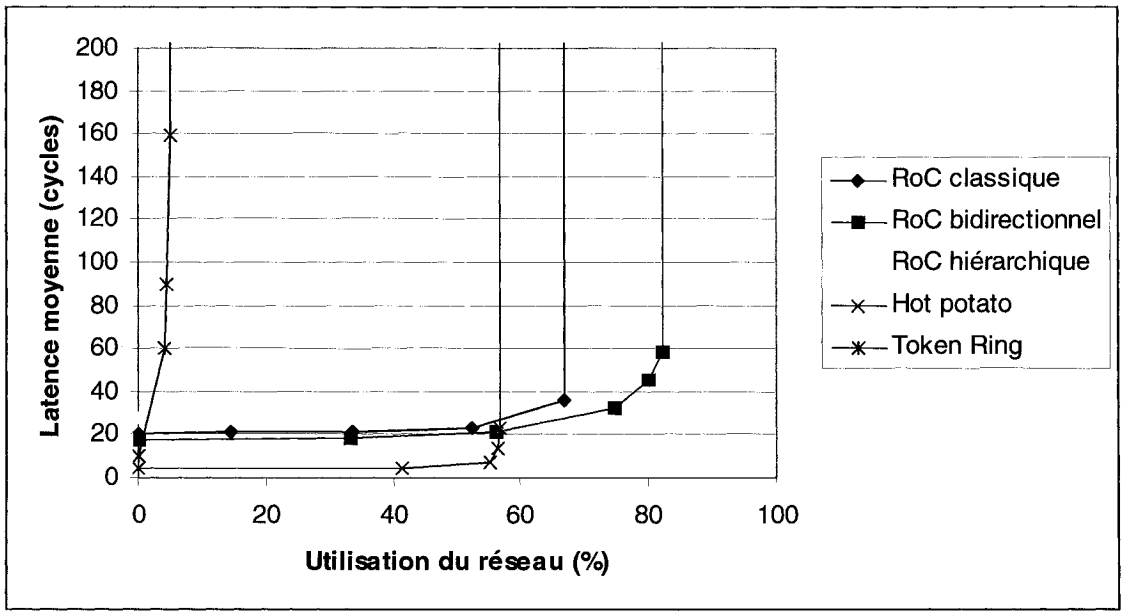


Figure 5.10 : Pourcentage d'utilisation de l'interface réseau (nœud)

Une conclusion intéressante est que le réseau en mailles peut satisfaire une application nécessitant une faible latence alors que la famille des RoC peut se révéler un choix intelligent pour une application roulant sur une puce où l'espace est restreint et où la

consommation de puissance est un enjeu important. D'autres réflexions portant sur ce sujet sont présentées dans la conclusion.

5.2.2. Trafic dirigé

Le trafic dirigé est un test qui vient mettre en évidence une faiblesse du RoC : le chemin unique. Peu importe la paire de ressources A et B dans tout le réseau, il n'existe qu'un seul chemin possible pour acheminer les paquets entre A et B. Quel que soit le type de RoC utilisé (standard, bidirectionnel ou hiérarchique), un seul chemin de transmission est possible. Dans un réseau à grande échelle, cette situation est rarissime en raison du risque de bris de liaison ou de panne de commutateur. Dans un système sur puce, ce risque est minime et l'accent ne doit pas être mis sur la tolérance aux fautes. La constatation majeure qui découle de ce test est qu'une source géographiquement proche de sa destination n'a jamais la chance d'envoyer de paquets, toujours selon le phénomène présenté à la section 4.3. Ainsi, la Figure 5.11 montre que les processeurs 10 à 15 peuvent envoyer des paquets à la mémoire 0 (slave 0), alors que les autres processeurs sont bloqués et ce même s'ils se retrouvent plus près dans la rotation (le processeur 1 étant le plus proche). Il est à noter que le même phénomène est observé pour le RoC bidirectionnel alors que les processeurs du milieu sont avantagés, puisque le voisin le plus éloigné de la mémoire 0 est le processeur 8. Par conséquent, les processeurs 6 à 10 ont un avantage.

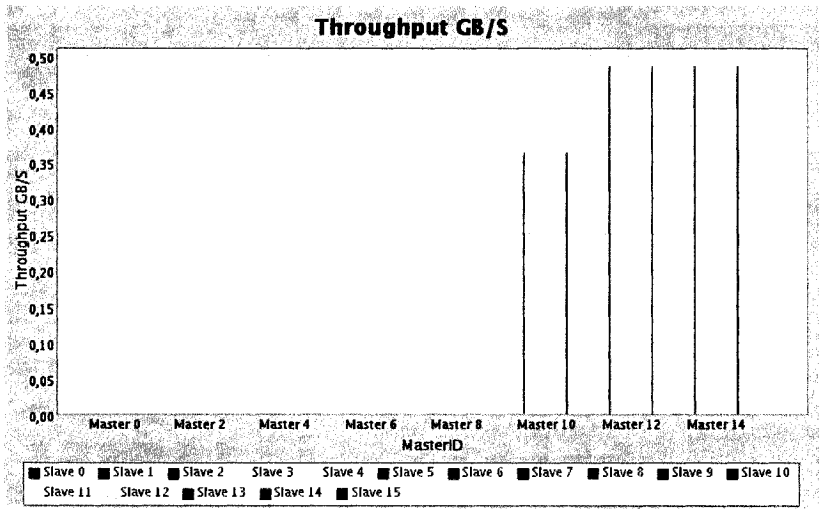


Figure 5.11 : Bande passante pour un RoC standard à 16 nœuds suivant un trafic dirigé (8 threads / processeur)

Ceci constitue donc une limitation par rapport au réseau en mailles utilisant un routage adaptatif comme c'est le cas pour Hot Potato (Figure 5.12). Étant donné que la ressource est accessible par quatre directions différentes et que les paquets sont délivrés selon une priorité qui augmente au fur et à mesure que le paquet demeure dans le réseau, toutes les ressources ont donc une chance relativement égale d'envoyer un paquet à une même destination. Il est à noter que l'algorithme de routage utilisé a une influence sur le comportement du système. Par exemple, avec une maille bidimensionnelle, les résultats seraient proches de ceux obtenus avec le RoC si un routage de type X-Y était utilisé puisqu'il existerait moins de chemins pour router les paquets. Cela revient toujours à un compromis entre la performance et le coût.

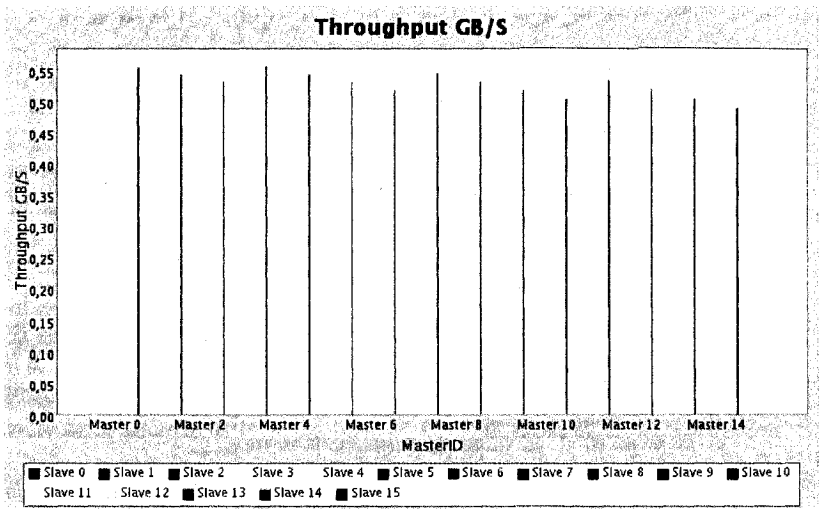


Figure 5.12: Bande passante pour un réseau Hot Potato à 16 nœuds suivant un trafic dirigé

Toutefois, pour le RoC, il est impossible de remédier à cette situation étant donné la nature même du réseau. Par contre, une chose est certaine : pour un réseau à N nœuds, il est toujours possible d'avoir N paquets simultanément en route pour une même destination, ce qui est considérablement meilleur que ce que peuvent offrir les topologies de bus et d'anneau.

5.2.3. Trafic voisin

Ce type de trafic montre jusqu'à quel point le RoC peut être considéré comme un réseau intégré sur puce très peu coûteux si certaines conditions sont respectées. Dans le cas où les communications se limitent à des transactions entre ressources voisines l'une de l'autre, l'utilisation du RoC bidirectionnel permet de sauver énormément d'espace, puisqu'un seul tampon sur 8 est occupé, en moyenne (voir Tableau 5.4). Les tampons forment le plus gros de l'espace consacré au NoC, un peu comme la mémoire occupe maintenant la majorité de la surface d'un SoC. S'il est possible de réduire de 85% le nombre de tampons requis par la spécification de base, le réseau ne constituera pas une énorme surcharge pour le système entier. Un tel trafic semble à priori utopique et irréaliste, mais il n'est pas rare dans la réalité d'observer des systèmes sur puce pour

lesquels une ressource communique majoritairement avec une poignée d'autres composants.

Tableau 5.3 : Taux d'occupation moyen des banques pour un RoC à 8 nœuds

NoC	Moyenne d'utilisation de tampons à saturation	Pourcentage
RoC	3,15	39.4%
RoC bitmap	3,91	48.9%
RoC bidirectionnel	0,98	12.3%
RoC hiérarchique	2,22	27.8%

Voilà pourquoi la connaissance et la maîtrise de l'application qui s'exécute sur le système dédié sont essentielles puisque cela permet au concepteur d'incorporer un réseau qui rend les communications performantes sans donner l'impression qu'il prive le développeur d'espaces utilisables. Il est à noter que les résultats découlant des simulations faites avec le *Hot Potato* ne sont pas discutés ici puisqu'ils n'apportent aucune information supplémentaire aidant à la comparaison entre les différents réseaux.

5.3. Simulations du RoC avec une application multimédia

Des simulations supplémentaires ont été effectuées pour évaluer les performances du réseau sur puce RoC lorsqu'il doit répondre à un trafic se rapprochant davantage de la réalité des systèmes sur puce. Différentes configurations du RoC ont donc été soumises au trafic découlant du modèle d'un encodeur MPEG4, modèle développé par STMicroelectronics. Cet encodeur transforme une image vidéo de format AVI en image vidéo de format MPEG. Le Tableau 5.4 donne la liste des modules nécessaires à l'encodage ainsi que le rôle qu'ils viennent jouer dans le traitement.

Tableau 5.4 : Description des modules de l'encodeur MPEG4

Classe	Description
Processeur ARM	Unité principale de traitement de l'encodage MPEG
Serveur DNS	Module associant un numéro d'identification à un composant
Module DCT	Module matériel dédié effectuant une transformée discrète en cosinus
Module SAD	Module matériel dédié calculant la différence entre les valeurs des pixels de deux macroblocs
Module de quantification	Module matériel dédié effectuant la quantification d'un macrobloc (opération requise par l'encodage MPEG)
Module VideoIn	Module dont la fonction est de lire le fichier d'entrée image par image à partir du disque dur et de placer ces images dans la mémoire de la plate-forme
Module VideoOut	Module dont la fonction est d'écrire le fichier de sortie sur le disque dur de la machine
Stack RAM	Pile rattachée à chacun des processeurs ARM
Text RAM	Mémoire contenant le programme à exécuter
Heap RAM	Section de mémoire (monceau) utilisée par les processeurs pour effectuer leurs calculs
Module de gestion de concurrence	Module permettant de créer des threads ainsi que de leur allouer l'exclusivité aux ressources du système
Module Horba	Module responsable de la répartition de certaines fonctions aux différents maîtres (matériels ou logiciels)

Par ailleurs, le Tableau 5.5 regroupe différents paramètres utilisés pour la simulation. Il est à noter que le nombre de threads matériels s'applique aux modules DCT, SAD, VideoOut, ainsi qu'au module de quantification.

Tableau 5.5 : Paramètres de simulation

Classe	Description
Nombre de processeurs	8
Nombre de threads par processeur	4
Taille de la pile / thread	4096 octets
Taille de la pile / processeur	16 Ko
Période de l'horloge des processeurs	5000 picosecondes
Fréquence du système	200 MHz
Taille de la Text RAM	384 Ko
Taille de la Heap RAM	1192 Ko
Nombre de threads matériels	32

Comme le montre la Figure 5.13, ce modèle d'encodeur MPEG4 utilise au total 26 ressources qui peuvent être réparties sur autant de nœuds. Ce modèle a été simulé selon trois configurations différentes du RoC. Les trois configurations utilisent la requête *bitmap* plutôt que la requête originale.

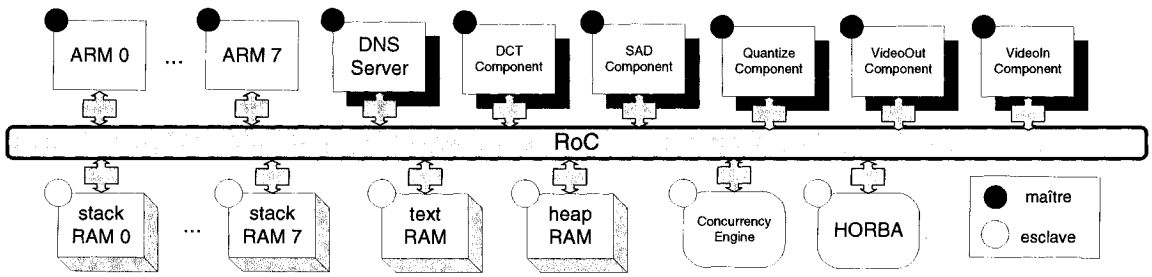


Figure 5.13 : Configuration à 26 nœuds

5.3.1. RoC classique / 26 nœuds

Une première configuration se veut être le RoC de base où un nœud dessert une seule ressource, qu'elle soit maître ou esclave. On retrouve donc 26 nœuds (se référer à la Figure 5.13 et au Tableau 5.5 pour les descriptions des modules) autour d'un rotateur central. Afin d'observer les effets du *mapping* sur les performances, tous les modules maîtres se suivent dans la rotation. Il en est de même pour tous les modules esclaves, tel qu'illustré à la Figure 5.14 (à noter le sens de rotation).

5.3.2. RoC bidirectionnel / 14 nœuds

Le RoC bidirectionnel permet bien sûr de réduire la latence en permettant au paquet de prendre le chemin le plus court entre sa source et sa destination. Cependant, le grand nombre de nœuds garde la latence relativement élevée. Une façon de réduire le nombre de nœuds dans le réseau est de jumeler un maître et un esclave. Le nœud (interface) doit agir en arbitre et acheminer un paquet au bon destinataire (maître ou esclave). Ceci est possible avec le protocole OCP puisqu'un des champs du paquet contient le type de

ressource qui l'a initié (maître ou esclave). Un examen sommaire du trafic créé par les ressources suggère d'emblée le jumelage entre les processeurs et leur pile. Cette association permet de diminuer le nombre de paquets injectés dans le réseau et ainsi de diminuer une possible contention. La Figure 5.15 résume cette configuration. Deux ressources se retrouvent seules puisqu'il y a 14 maîtres comparativement à 12 esclaves.

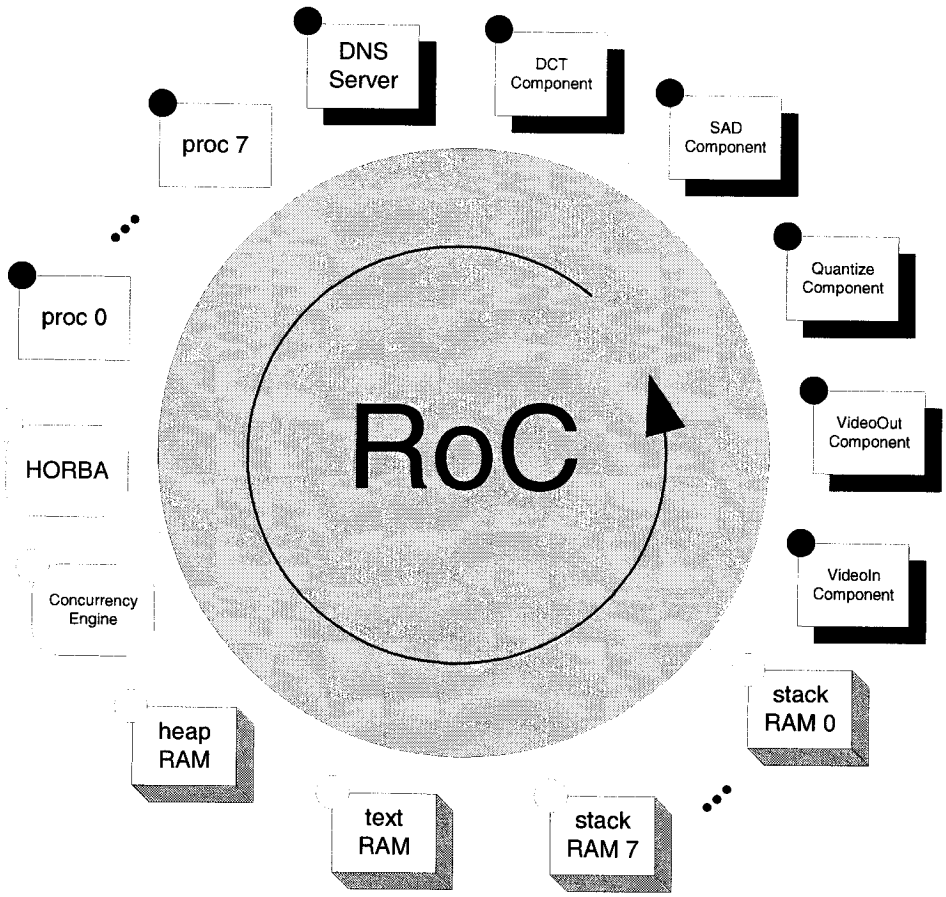


Figure 5.14 : Ordre des ressources dans la rotation

5.3.3. RoC hiérarchique / 14 nœuds

Le RoC hiérarchique utilise la même distribution maître/esclave associée au nœud. Pour cette configuration, deux satellites de 7 nœuds sont utilisés. Rappelons qu'un rotateur central permet de relier entre eux les satellites externes.

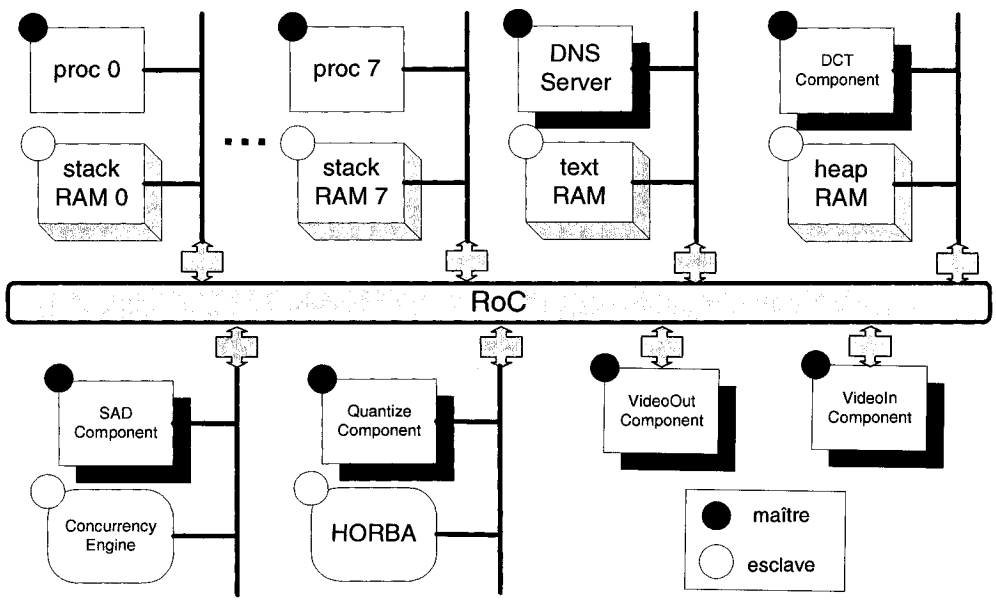


Figure 5.15 : Configuration à 14 nœuds

5.3.4. Résultats obtenus

Les résultats ont été pris après l'encodage d'une seule image (le fichier en comprenant 30, formant ainsi une séquence vidéo d'une durée d'une seconde). En examinant la nature des transactions entre les différents modules, on constate que ce type de trafic n'avantage pas nécessairement le RoC. En effet, la Figure 5.16 montre que la plupart des maîtres initient des transactions à deux esclaves en particulier: l'esclave #9 (Heap RAM) et l'esclave 11 (Horba). Ainsi, cette situation tend à se comparer au trafic de type *même endroit* (section 5.1.2.2) où tous les paquets sont dirigés au même endroit. Ce type de trafic mène à une hausse de la contention pour ces deux esclaves. Les simulations de la version de base du RoC montrent en effet que jusqu'à 30 paquets à la fois doivent se diriger à la mémoire. Ce nombre augmente à 55 en ce qui attrait au Horba.

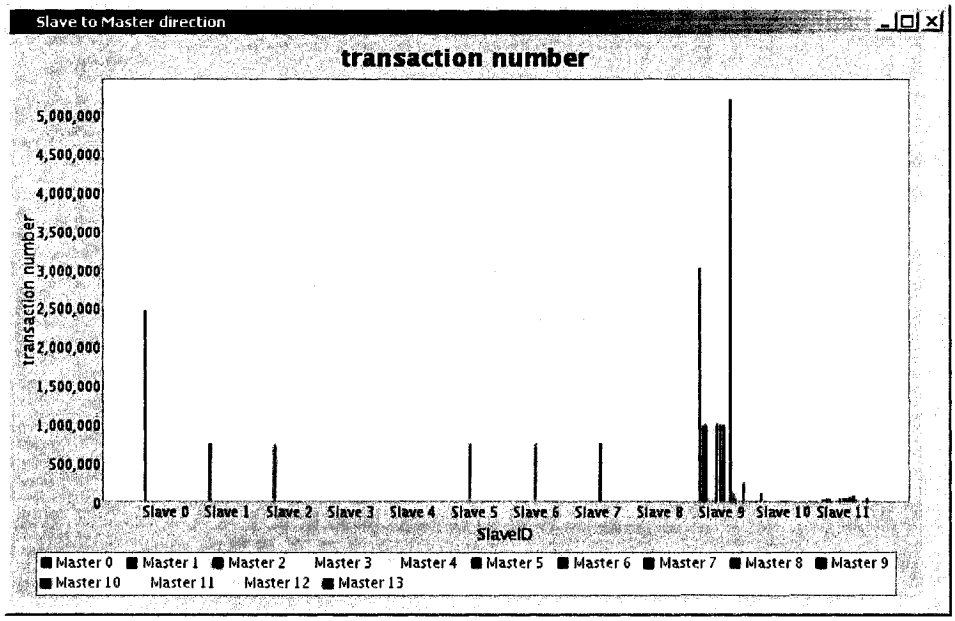


Figure 5.16 : Nombre de transactions traitées par les esclaves (réponses)

La Figure 5.17, la Figure 5.18 et la Figure 5.19 montrent les délais moyens obtenus pour l'envoi d'un paquet d'un maître à un esclave en utilisant le RoC classique à 26 nœuds, le RoC bidirectionnel à 14 nœuds et le RoC hiérarchique à 14 nœuds, respectivement. On constate à la Figure 5.18 que la latence moyenne entre le maître 7 (processeur ARM) et l'esclave 9 (Heap RAM) est très élevée, comparativement aux autres latences. Ce phénomène découle du fait qu'à un certain moment de la simulation, tous les processeurs communiquent en même temps avec la mémoire. Or, l'utilisation du RoC bidirectionnel fait en sorte que, pour ce *mapping*, les processeurs 2 à 7 utilisent des banques allant dans le sens horaire, alors que les processeurs 0 et 1 ainsi que les autres maîtres (excepté le 8) utilisent des banques allant dans le sens anti-horaire. Ainsi, le processeur 7 a de très fortes chances de se buter à un tampon plein étant donné que cinq autres processeurs ont accès aux banques avant lui. Ce phénomène ne se produit pas pour le RoC classique puisque tous les composants ont accès à l'ensemble des banques, et non à la moitié. Une façon de diminuer l'ampleur de ce problème serait de changer la disposition des ressources, en insérant la mémoire Heap entre les processeurs #3 et #4, par exemple.

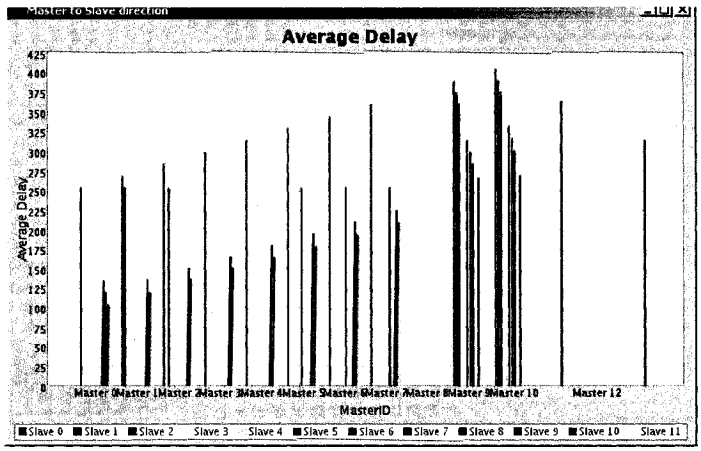


Figure 5.17 : Latence moyenne des paquets (en ns) en utilisant le RoC classique

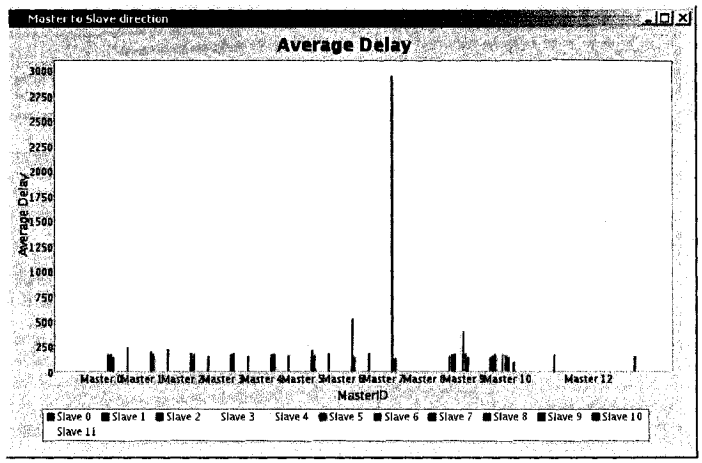


Figure 5.18 : Latence moyenne des paquets (en ns) en utilisant le RoC bidirectionnel

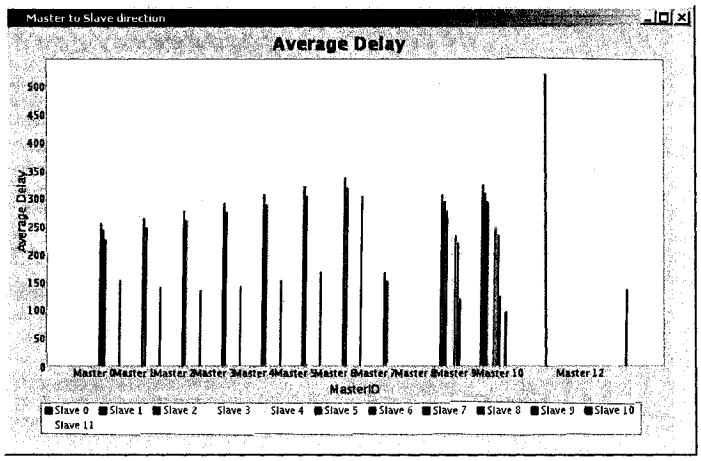


Figure 5.19 : Latence moyenne des paquets (en ns) en utilisant le RoC hiérarchique

Par ailleurs, pour ce type de trafic, on ne note pas de véritable amélioration sur la latence lorsqu'on passe d'un modèle à un autre. Ceci est encore dû à la forte contention qui garde les paquets « prisonniers » sur les nœuds pendant plusieurs rotations. À cet effet, le Tableau 5.6 présente le délai maximal observé pour une transaction (maître-esclave ou esclave-maître). Encore une fois, pour le RoC bidirectionnel, la situation présentée ci-haut conduit à un délai maximal de 2 550 000 ns, soit l'équivalent de 170 000 étapes (rotations). Un paquet est donc prisonnier sur un nœud pendant l'équivalent de plus de 12 000 tours complets! Si l'encodage doit être effectué en temps réel, cette situation est tout simplement catastrophique, même si les contraintes de temps sont assouplies. Par conséquent, l'analyse du trafic relié à l'application ainsi que la disposition des composants du système sont deux critères primordiaux qui permettront d'utiliser efficacement un réseau intégré sur puce comme le RoC.

Tableau 5.6 : Latence maximale observée avec le trafic MPEG4 selon différentes configurations

Classe	Latence maximale (ns)	Rotations équivalentes
RoC classique, 26 nœuds	6100	407
RoC bidirectionnel, 14 nœuds	2 550 000	170 000
RoC hiérarchique, 26 nœuds	7500	500

À la lumière des résultats obtenus, on peut conclure que le RoC est un réseau qui s'inscrit à merveille dans le contexte de la réutilisation étant donné le taux d'utilisation élevé de ses composants. Il est facilement extensible et peu coûteux en matériel, dans la mesure où certaines optimisations sont employées. Le désavantage principal de ce réseau est sa latence, proportionnelle au nombre de composants qui y sont connectés. Cependant, dans le contexte d'un système dominé par des données, la latence importe peu, l'accent étant mis sur le débit. Par contre, pour un système dominé par du contrôle où des contraintes de temps réel dures peuvent s'appliquer, une latence élevée peut constituer un problème.

Conclusion et travaux futurs

L'étude des systèmes intégrés sur puce a permis de mettre en évidence quelques unes de leurs lacunes, en particulier les problèmes découlant des architectures actuelles de communication. En approfondissant le domaine récent des réseaux sur puce et en intégrant les concepts de réseautique sur des systèmes à plus petite échelle, l'idée est venue de concevoir un NoC qui utilise efficacement les ressources limitées dont il dispose. Le RoC se veut un réseau sur puce facilement extensible qui, lorsque optimisé, consomme peu de ressources tout en offrant une bande passante respectable et une latence relativement basse. Le RoC n'est pas conçu pour optimiser la latence. D'ailleurs, un réseau en mailles comme le Hot Potato et même la plupart des architectures de bus effectuent les transactions en un temps moindre en absence de contention. L'idée derrière le RoC est de mettre l'accent sur la réutilisation maximale des ressources. Tel qu'expliqué dans le chapitre 1, le développement rapide d'un système sur puce repose sur la réutilisation des blocs IP préalablement conçus. Cette idée peut également être perçue d'un autre angle : celui d'une utilisation optimale de ces blocs. Le RoC rencontre cet objectif, puisque la topologie et le routage engendrent un très haut taux d'utilisation de ses composants (jusqu'à 80% pour un RoC bidirectionnel). Cette caractéristique est un atout pour le développement de systèmes performants où l'espace disponible est limité. Il est plus avantageux de privilégier un design où 10 blocs opèrent à 100% plutôt que 20 à 50%.

Pour valider la fonctionnalité du NoC développé dans le cadre de ce projet, le RoC a été soumis à plusieurs types de trafic. Les différentes simulations, que ce soit celles correspondant à un trafic aléatoire ou celles rattachés à l'application MPEG4, ont permis de mieux cerner les possibilités du NoC de même que ses limitations. Les performances du RoC dépendent énormément de la nature du trafic (distribution des destinations) ainsi que de l'ordre dans lesquels les composants sont disposés sur la puce. Le développeur a

donc tout à gagner à effectuer une analyse approfondie de l'application, étant donné que la plupart des SoC sont dédiés à une tâche bien précise.

La contribution majeure de ce travail est le développement du modèle du RoC à haut niveau et l'intégration de ce modèle dans une plate-forme d'exploration architecutrale. Le modèle a haut niveau a permis de valider l'algorithme et la fonctionnalité des composants du NoC alors que son intégration a permis de caractériser ses performances et de le comparer avec d'autres topologies. Les résultats prometteurs ouvrent la voie à de grandes possiblités au niveau du traitement des applications de type *stream*. En plus de s'inscrire parfaitement dans le contexte de la réutilisation grâce à un taux d'utilisation élevé de ses composants, le RoC cherche à diminuer la surcharge en espace et en puissance qui accompagne l'ajout d'une architecture de communication structurée.

Considérations pour le futur

Le monde des réseaux intégrés sur puce en est encore à ses balbutiements. Plusieurs universités et compagnies développent leurs propres topologies en espérant trouver la configuration idéale. On en est donc encore à l'étape de l'expérimentation et des découvertes. D'ici les cinq prochaines années, il sera primordial de développer des normes reliées aux NoC pour faire converger les recherches sur quelques aspects des réseaux. Une classification des NoC selon les applications et les ressources matérielles disponibles permettra de proposer des systèmes multiprocesseurs complets incluant le réseau sur lequel les processeurs communiqueront.

Par ailleurs, il faut garder en tête que les réseaux intégrés sur puce ne constituent pas la solution pour tous les problèmes reliés aux communications sur puce. Les NoC apportent aussi quelques problèmes avec eux.

On s'attend à ce que les NoC prennent une part notable du budget alloué à l'aire des SoC, en raison de la complexité grandissante des algorithmes de routage et des politiques de gestion des transactions, qui affectent le chemin de contrôle des interconnexions. Par ailleurs, l'ensemble des commutateurs en charge de réaliser l'acheminement à haute vitesse des paquets requiert une fraction significative de l'aire disponible [BOZZ04]. Le problème relié à l'optimisation de la taille des tampons des commutateurs demeure entier. En effet, rejeter un paquet ou le router inadéquatement en raison d'une taille de tampon inappropriée réduit les performances et augmente la consommation de puissance sur la puce [VAMA02].

En général, le développement d'un SoC implique l'intégration de plusieurs blocs spécialisés issus de différentes technologies. Ainsi, on se retrouve avec plusieurs noyaux (*core*) ayant des tailles, des fonctionnalités et des requis de communications différents (voir Figure 1 b)). Par conséquent, il peut être impossible d'employer une structure régulière, comme celle illustrée à la Figure 1 a). De plus, la bande passante requise par les différents blocs peut varier énormément, certains d'entre eux n'ayant même aucun besoin de communiquer. En utilisant un tel système jumelé à une architecture en mailles, par exemple, on peut constater une sous-utilisation des liens et des commutateurs à un endroit ou une congestion locale à un autre endroit. Ces facteurs motivent l'utilisation d'un réseau spécifique à une seule application [BEBE04].

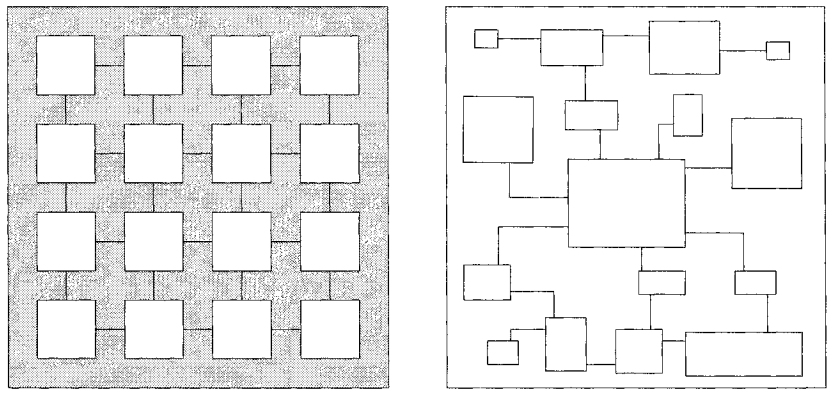


Figure 1 : Homogénéité de la topologie versus hétérogénéité des blocs et des communications

Malgré l'émergence des NoC, une place de choix est encore disponible pour les architectures de bus. Ces architectures peuvent en effet servir de premier niveau de communication entre quelques blocs, le trafic restant étant acheminé à un deuxième niveau de communication, le NoC proprement dit. La Figure 2 montre en effet qu'un maître (M) qui communique fréquemment avec quelques esclaves (E) peut utiliser un bus pour le faire. Le trafic destiné à d'autres blocs est acheminé au NoC via un pont (P). Les raisons pour utiliser cette configuration sont multiples. D'une part, le bus permet une meilleure utilisation de la bande passante offerte puisqu'un même lien est partagé par plusieurs blocs ayant des requis différents. D'autre part, les interfaces réseau sont plus coûteuses en espace qu'un adaptateur de bus. En changeant les interfaces pour des adaptateurs, le coût global en espace occupé par le réseau de communication diminue.

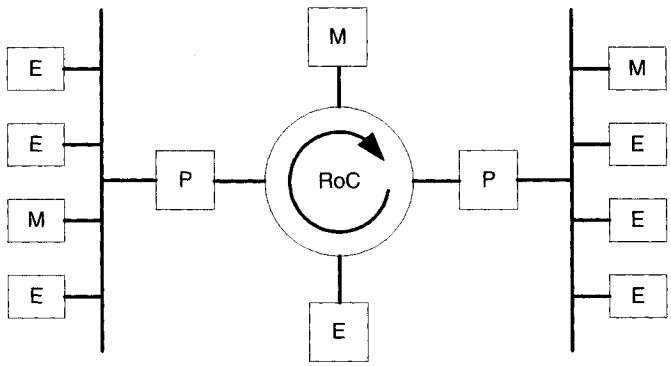


Figure 2 : Utilisation d'un NoC conjointement à une architecture de bus

Finalement, le nombre de commutateurs est réduit significativement lorsque les bus sont utilisés. Cela est un avantage puisque les commutateurs sont plus gourmands en espace que les bus, étant donné les files de paquets et les algorithmes plus complexes que leur utilisation nécessite [WIGO02].

Travaux futurs

À la lumière de ces considérations, les travaux qui pourraient découler de ce projet sont nombreux. Le modèle initial du RoC (ainsi que ses variantes) néglige toute forme de contrôle de flot. Cette situation mène à un indéterminisme qui proscrit l'utilisation du RoC lorsque des contraintes de temps réel sont en jeu. Un travail intéressant à effectuer serait donc d'étendre le réseau à l'ensemble des domaines d'applications en ajoutant des caractéristiques reliées au contrôle de flot. Une première caractéristique serait de supporter la réservation de tampons pour une utilisation exclusive. Par exemple, le nœud #0 aurait l'exclusivité des tampons associés à la destination #2 le temps qu'il envoie un *stream* de données. Cette caractéristique permettrait une meilleure maîtrise du mode en rafale (*burst*) présenté à la section 4.2.

Une deuxième caractéristique serait de contrôler la rotation des banques de façon individuelle. La Figure 3 présente une situation où un trafic jugé plus prioritaire arrive à

sa destination, le nœud #1 d'un RoC à 4 ressources. La rotation des tampons associés à la destination 1 pourrait être interrompue pour donner le temps à la ressource #1 d'accepter les paquets provenant de l'extérieur. Toutefois, les rotations des tampons associés aux destinations 0, 2 et 3 pourraient continuer de façon à permettre à ces trois ressources de communiquer entre eux.

Cette caractéristique sous-entend que le RoC supporte des trafics de priorités différentes, ce qui n'est pas le cas actuellement. L'algorithme utilisé présentement est basé sur le tourniquet, qui donne une chance égale à toutes les destinations. L'ajout d'une priorité dans le processus de décision se rapprocherait davantage de la réalité des systèmes sur puce et des systèmes embarqués temps réel. Un système de priorités dynamiques pourrait également être instauré pour éviter la famine d'un trafic toujours interrompu par des transactions plus prioritaires.

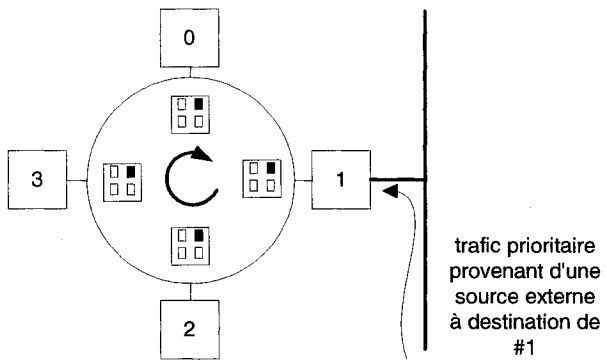


Figure 3 : Exemple de contrôle de flot

Travaux reliés

Deux projets reliés au RoC ont été réalisés ou sont en voie de l'être. Un premier projet consiste à modéliser le RoC à plus bas niveau afin de le caractériser sur l'espace réel qu'il peut occuper, la fréquence d'opération qu'il peut atteindre ainsi que la puissance qu'il

dissipe. Ce projet est en cours de réalisation et les résultats seront éventuellement disponibles dans [STPI05].

Un deuxième terminé en avril 2005 a été la modélisation en SystemC du HyRoC (*Hyper Ring on Chip*) imaginé par la compagnie STMicroelectronics. Le HyRoC, qui est illustré à la Figure 4, se veut un réseau bidimensionnel de nœuds, chaque nœud étant connecté à 4 rotateurs RoC. De ces quatre roues (dans la nomenclature HyRoC), deux sont horizontales et les deux autres sont verticales. C'est comme si un nœud était connecté à deux RoC bidirectionnels, à l'exception que le nœud peut transiger avec 4 banques à la fois, une par roue. Le modèle initial du HyRoC permet donc à un nœud de recevoir ou d'envoyer plusieurs paquets à la fois, ce qui n'est pas possible avec le RoC. La topologie en maille permet d'utiliser le routage XY pour s'assurer de l'ordre d'arrivée des paquets. Avec une configuration de huit maîtres et huit esclaves (4 x 4) et un trafic aléatoire, les résultats préliminaires montrent que les délais restent stables même lorsqu'on augmente le nombre de threads par processeur et le HyRoC offre une bande passante de 17 Gb/s, comparativement à 14 Gb/s pour une même configuration.

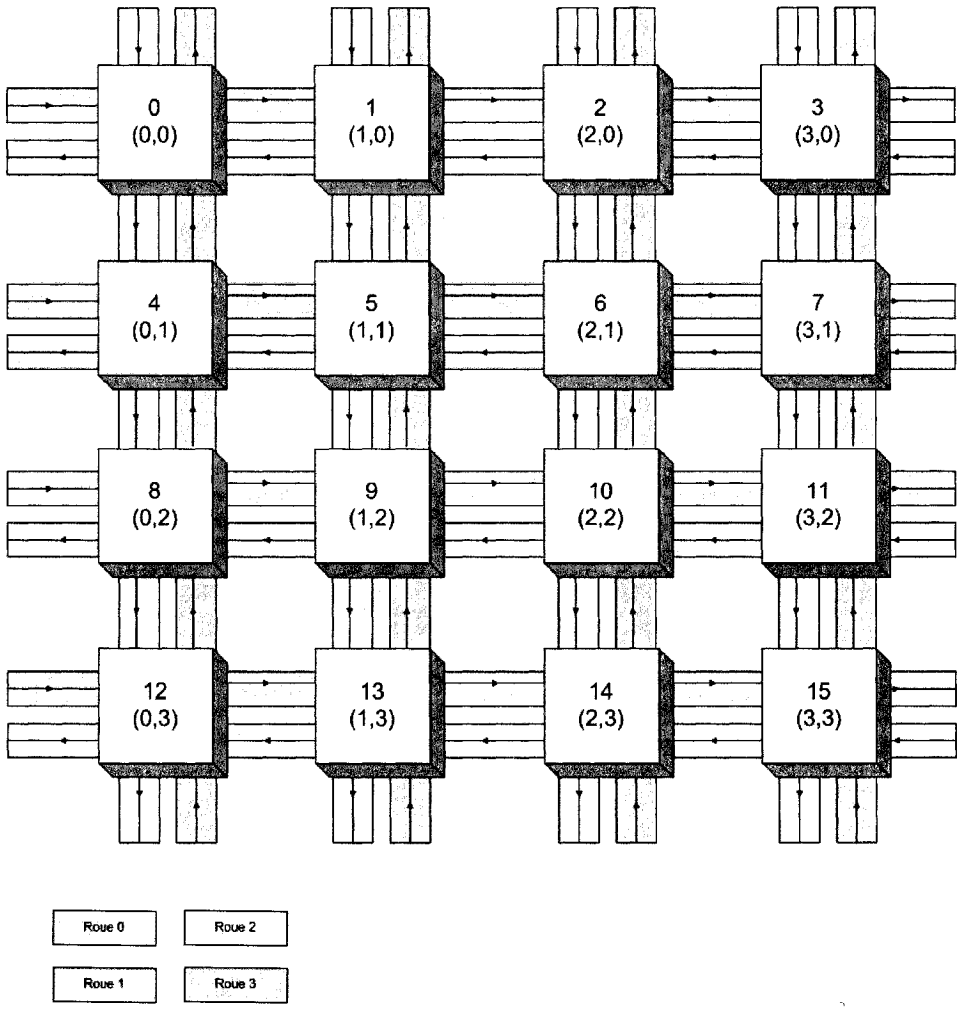


Figure 4 : Vue d'ensemble du HyRoC (Hyper Ring-on-Chip)

Les NoC en sont encore à leurs débuts et les recherches des prochaines années permettront de mieux les caractériser de façon à en tirer le meilleur profit.

Références

[ACGM03] ADRIAHANTENAINA, A., CHARLERY, H., GREINER, A., MORTIEZ, L. 2003. "SPIN: a Scalable, Packet Switched, On-Chip Micro-network", *Design, Automation and Test in Europe Conference and Exhibition*, IEEE, p. 70-73

[ARM01] ARM. 2001. *AMBA Home Page.* [en ligne]. http://www.arm.com/products/solutions/AMBAHomePage.html (Page consultée le 25 février 2005)

[AYKJ04] ANJO, K., YAMADA, Y., KOIBUCHI, M., JOURAKU, A. 2004. "BLACK-BUS: A New Data-Transfer Technique using Local Address on Networks-on-Chips". *18th International Parallel and Distributed Processing Symposium*, IEEE, 10 p.

[BEBE04] BERTOZZI, D., BENINI, L. 2004. "Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-chip", *IEEE circuits and systems magazine*, p. 18-31

[BEDE02] BENINI, L., DE MICHELI, G. 2002. "Networks on Chips: A New SoC Paradigm", *Computer*, IEEE, p. 70-78

[BEDE02b] BENINI, L., DE MICHELI, G. 2002. "Networks on Chips: A New Paradigm for *Systems on Chip* Design", *Design, Automation and Test in Europe Conference and Exhibition*, IEEE, p. 418-419

[BERT03] BERTOLA M. 2003. *Conception, réalisation et étude d'une plate-forme générique basée sur le protocole AMBA AHB*, Mémoire de maîtrise, Département de génie électrique, École Polytechnique de Montréal.

[BOZZ04] BONA, A., ZACCARIA, V., ZAFALON, R. 2004. "System Level Power Modeling and Simulation of High-End Industrial Network-on-Chip". *Design, Automation and Test in Europe Conference and Exhibition Designers' Forum (DATE04)*, IEEE, p. 318-323 Vol.3

[CARN00] CARTER, N. 2000. *Networks II – Trees and Butterflies*, chapitre du cours ECE412: Advanced Computer Architecture, 15 p.

[CBRB04] CHEVALIER, J., BENNY, O., RONDONNEAU, M., BOIS, G., ABOULHAMID, E. M., and BOYER, F.-R. 2004. "SPACE: A Hardware/Software SystemC Modeling Platform Including an RTOS". *Design & Verification Conference (DVCon).* San Jose, USA.

[CHPA04] CHAN, J., PARAMESWARAN, S. 2004. "NoCGEN: A Template Based Resue Methodology for Networks on Chip Architecture", *17th International Conference on VLSI Design*, IEEE, p. 717-720

[CLEA01] Voir le site Web de Clearspeed™ : http://www.clearspeed.com

[CLEA03] CLEARSPEED™. 2003. *ClearConnect® Bus : High Performance On-Chip Interconnect*, description.

[DATO01] DALLY, W., TOWLES, B. 2001. "Route Packets, Not Wires: On-Chip Interconnection Networks". *Design Automation Conference (DAC).* IEEE. p. 684-689

[DELI05] DELISLE, K. 2005. *Modélisation d'un réseau intégré sur puce et intégration sur une plate-forme d'exploration architecturale*, Rapport de projet de fin d'études, École Polytechnique de Montréal, 67 p.

[FILI02] FILION L. 2002. *Analyse, implantation et intégration d'une bibliothèque pour la spécification des sytèmes embarqués dans une méthodologie de codesign*, Mémoire de maîtrise, Département de génie électrique, École Polytechnique de Montréal.

[FOAL03] FOALENG J. 2003. *Modélisation et Outils d'analyse de Performances de réseaux intégrés sur puce,* Rapport de stage, STMicroelectronics, 44 p.

[FORS02] FORSELL, M. 2002. "A Scalable High-Performance Computing Solution for networks on Chips". *Micro,* IEEE, Volume 22, Numéro 5, p. 46-55

[GPIS03] GRECU, C., PANDE, P., IVANOV, A., SALEH, R. 2003. *Network On Chip – A New Interconnect Architecture for SoCs.* 29 p.

[GPIS03b] GRECU, C., PANDE, P., IVANOV, A., SALEH, R. 2003. "Design of a Switch for network on Chip Applications", *2003 International Symposium on Circuits and Systems*, IEEE, p. 217-220

[HEWC04] HENKEL, J., WOLF, W., CHAKRADHAR, S. 2004. "On-chip networks: A scalable, communication-centric embedded system design paradigm", *17th International Conference on VLSI Design*, IEEE, p. 845-851

[IBM03] IBM. 2003. *CoreConnect™ bus architecture.* [en ligne]. http://www-03.ibm.com/chips/products/coreconnect/ (Page consultée le 24 février 2005)

[ITRS04] Voir le site Web de l'ITRS : http://public.itrs.net

[JANA01] JANTSCH A. 2001. "Introduction to networks on chip", *Workshop at the European Solid-State Circuit Conference (ESSCIRC)*, 15 p.

[KJSF02] KUMAR, S., JANTSCH, A., SOININEN, J., FORSELL, M. 2002. "A Network on Chip Architecture and Design Methodology". *IEEE Computer Society Annual Symposium on VLSI,* IEEE. p. 105-112

[MAMA04] MAS, G., MARTIN, P. 2004. "Network-on-chip: the intelligence is in the wire", *IEEE International Conference on Computer Design*, IEEE, p. 174-177

[MNTK04] MILLBERG, M., NILSSON, E., THID, R., KUMAR, S. 2004, "The Nostrum backbone-a communication protocol stack for Networks on Chip". *17th International Conference on VLSI Design*, IEEE, p. 693-696

[NILS02] NILSSON, E. 2002. *Design and Implementation of a hot-potato Switch in a Network on Chip*, Mémoire de maîtrise, Laboratoire d'électronique et de systèmes informatiques, Institut royal de technologie, Kista, Suède

[OSCI01] Voir le site Web de SystemC : http://www.systemc.org

[OSCI03] OSCI. 2003. *SystemC Version 2.0.1 User's guide.* [en ligne]. http://www.systemc.org (Page consultée le 29 octobre 2002)

[PAPB02] PAULIN, P. G., PILKINGTON, C., BENSOUDANE, E. 2002. "StepNP: a system-level exploration platform for network processors". *Design & Test of Computers,* IEEE. 19:6. p. 17-26.

[RGRM03] RIJPKEMA, E., GOOSSENS, K., RADULESCU, A., MEERBERGEN, J. 2003. "Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip". *Design Automation and test in Europe*, IEEE, p. 350-355

[ROVF05] ROSTISLAV, D., VISHNYAKOV, V., FRIEDMAN, E. 2005. "An Asynchronous Router for Multiple Service Levels Networks on Chip". *11th IEEE International Symposium on Asynchronous Circuits and Systems*, IEEE, p. 44-53

[SAAN03] SAASTAMOINEN, I., ALHO, M., NURMI, J. 2003. "Buffer Implementation for Proteo Networks-on-Chip", *2003 International Symposium on Circuits and Systems*, IEEE, p. 1-6

[SIBR04] SIEBENBORN, A., BRINGMANN, O., ROSENSTIEL, W. 2004. "Communication Analysis for Network-on-Chip Design". *International Conference on Parallel Computing in Electrical Engineering (PARELEC)*, IEEE, p. 315-320

[SILI02] Silicore. 2002. *Wishbone System-On-Chip (SoC) Interconnection Architecture for Portable IP Cores*, révision B.3, disponible en-ligne à http://www.opencores.org/projects.cgi/web/wishbone/wishbone (Page consultée le 12 décembre 2004)

[SONI01] Sonics. 2001. *SiliconeBackplane™ III.* [en ligne]. http://www.sonicsinc.com/sonics/products/siliconbackplaneIII/ (Page consultée le 8 janvier 2005)

[STMI00] Voir le site Web de STMicroelectronics : http://www.st.com/

[STMI03] STMicroelectronics. 2003. *STBus Communication System Concepts and Definitions*. Spécifications disponibles au http://www.stmcu.com/inchtml-pages-STBus_intro.html

[STPI05] ST-PIERRE, Francis. 2005. Mémoire. *Modélisation à bas niveau et analyse d'un réseau intégré sur puce*. École Polytechnique de Montréal. (mémoire en préparation)

[SYNO03] Voir le site Web de Synopsys : http://www.synopsys.com/

[USSEO1] USSELMANN, R. 2001. *OpenCores SoC Bus Review*, Rev. 1.0, 12 p.

[VAMA02] VARATKAR, G., MARCULESCU, R. 2002. "Traffic Analysis for On-chip Networks Design of Multimedia Applications". *Design Automation Conference*, IEEE. p. 795-800

[VSIA00] Voir le site Web de l'alliance VSI : http://www.vsi.org/

[WIGO02] WIELAGE, P., GOOSSENS, K. 2002. "Networks on Silicon: Blessing or Nightmare? ". *Euromicro Symposium on Digital System Design*, IEEE. p. 196-200

[WILI00] WIKLUND, D., LIU, D. 2000. "Switched Interconnect for System-on-a-Chip Designs", *IP2000 Europe Conference*, p. 187-192

[WILI03] WIKLUND, D., LIU, D. 2003. "SoCBUS: Switched Network on Chip for Hard Real Time Embedded Systems". *International Parallel and Distributed Processing Symposium*, IEEE, 8 p.

[WING01] WINGARD, D. 2001. "MicroNetwork-Based Integration for SOCs". *Design Automation Conference (DAC)*, IEEE, p. 673-677

[YOO03] YOO, S. 2003. "On-Chip Communication Design", *DATE Master Course*, 67 p.

[ZESU03] ZEFERINO, C., SUSIN, A. 2003. "SoCIN: A Parametric and Scalable Network-on-Chip". *16th Symposium on Integrated Circuits and Systems Design*, IEEE, p. 169-174

Annexes

ANNEXE A
Fichier d'intégration du RoC classique à la plate-forme StepNP

Cette section contient le code du fichier SOCPFuncROC.h qui se veut le pont entre le protocole OCP et le RoC.

```
#include <stdlib.h>
#include <iostream.h>
#include <stream.h>
#include <string>
#include <systemc.h>
#include <component/interconnect/include/packet.h>
#include <component/interconnect/include/socp.h>
#include <component/interconnect/include/bitmap.h>
#include "bank.h"
#include <component/interconnect/include/node.h>
#include "circularChannel.h"
#include <component/interconnect/instrumentation/perNoc.h>
#include <component/interconnect/include/socpChannelBase.h>

#define MASTERS 14
#define SLAVES 12
#define MAXTHREAD 32


typedef uint AddrT;
typedef uint  DataT;

template <class AddrT, class DataT, int NB_NODES = 8, int NB_MASTERS=8,
int NB_SLAVES = 8, int MAX_MASTERS = 32, int MAX_SLAVES = 32>
    class SocpFuncROC:
public SocpChannelBase<AddrT, DataT, MAX_MASTERS, MAX_SLAVES> {

      private:
      typedef  SocpInfo<AddrT, DataT> Info;
      typedef Buffer<AddrT, DataT> buffer;
      typedef Packet<AddrT, DataT> packet;
      typedef Bitmap<NB_NODES> bitmap;
      typedef  Node<AddrT, DataT, NB_NODES> node;
      typedef  Bank<AddrT, DataT, NB_NODES> bank;
      typedef  CircularChannel<AddrT, DataT, NB_NODES> circular_connect;
      typedef  sc_fifo<packet*> fifo;

      public:
      int numReq;
      sc_buffer<packet> resPack[NB_NODES*2];
      int i,j;

#   define MAX_ILEAVE   8
```

```
      struct IlI {

      AddrT a1, a2;           // -- address range of interleave
      int startSlave;         // -- slave where we start the interleave
      int numSlave;           // -- how many slaves we interleave over
      int offsetBits;         // -- start bit # in addr for interleave

      IlI() {
        a1 = a2 = 0;
        startSlave = -1;
        offsetBits = 0;
      }

      } il[MAX_ILEAVE+1];

      int numIL;

      long compteur, compteur2;
      node* nodes[NB_NODES];
      bank* banks[NB_NODES];
      circular_connect* core;
      packet* input;
      packet* output;

      sg_in<bool> clk;

      Info* start[NB_NODES][NB_NODES][MAXTHREAD];
      bool                          flag[NB_NODES][NB_NODES][MAXTHREAD];
// burst from that source, a second dimension is added in case of
multithreaded source
      DataT* dataPtr[NB_NODES][NB_NODES][MAXTHREAD];                    //
burst value
      unsigned int cnt[NB_NODES][NB_NODES][MAXTHREAD];

      long getCycle() {return core->getCycle();}
      double getBufferUsed() {return banks[0]->getBufferUsed();}

      void dynamic_run() {

        while(1) {

            sc_spawn(sc_bind(&SocpFuncROC::fifo_in2output_buffer,
this));
            sc_spawn(sc_bind(&SocpFuncROC::input_buffer2fifo_out,
this));
            sc_spawn(sc_bind(&SocpFuncROC::fifo_out2destination, this));
            wait(3);
        }
      }

      void fifo_in2output_buffer()
      {
        packet* pack;
        int s, d;

        for(i = 0; i < NB_NODES; i++)
        for(j = 0; j < NB_NODES; j++) {
            if (nodes[i] -> output_buffer[j] ->isFull() != 1)
            {
              if (nodes[i] -> fifo_in[j]  -> nb_read(pack)) {
```

```
                    //debugging
                    s = pack -> getSource();
                    d = pack -> getDestination();
                    if ((s != i) or (d != j)) cout <<  "DANGER: mixmatch
in fifo_in" << endl;
#ifdef ROC_DEBUG1
                    cout << " fifo_in2ouput_buffer: (sour -> dest): " << s
<< " -> " <<  d << endl << endl;
#endif


                    nodes[i] -> output_buffer[j]->setPacket(*pack);
                    delete pack;
                }
            }
        }
    }

    void input_buffer2fifo_out()
    {

      for(i = 0; i < NB_NODES; i++)
      for(j = 0; j < NB_NODES; j++) {
          if (nodes[i] -> input_buffer[j] -> isFull() == 1)
          {
                input = new packet(nodes[i] -> input_buffer[j] ->
getPacket());
                if (nodes[i] -> fifo_out[j] -> nb_write(input)) {
                nodes[i] -> input_buffer[j] -> empty();
#ifdef ROC_DEBUG1
                cout << " input_buffer2fifo_out: packet transfered in
fifo_out (sour -> dest): " << j  << " -> " <<  i << endl << endl;
#endif
                }
                else {
                delete input;
//#ifdef ROC_DEBUG1
                cout << "input_buffer2fifo_out: fifo_out is full:   "
<< j << "   "<< i  << endl;
//#endif
                }

          }
      }
    }

    void fifo_out2destination()
    {

      packet* pack;
      int s, d, t;

      for(i = 0; i < NB_NODES; i++)
      for(j=0; j < NB_NODES; j++)
      {
          if(nodes[i] -> fifo_out[j] -> nb_read(pack)) {

            //debugging
            s = pack -> getSource();
            d = pack -> getDestination();
```

```
            if ((s != j) or (d != i)) cout <<  "DANGER: mixmatch in
fifo_out" << endl;
#ifdef ROC_DEBUG1
            cout << "fifo_out2destination (sour -> dest): " << s  << "
-> " <<  d  << endl << endl;
#endif

            Info* p = pack -> getPayload();
            t = p->mThreadID;

            // if burst transaction and (read response from slave or
write from master)
            if( (p->length > 1) && ( (p->mCmd == Info::RD && p-> sResp
!= Info::SNULL) || (p->mCmd == Info::WR && p-> sResp == Info::SNULL) ) )
            {
                // we first create a data array
                if(!flag[s][d][t])
                {
                  flag[s][d][t] = true;
                  if(dataPtr[s][d][t])
                      delete dataPtr[s][d][t];
                  dataPtr[s][d][t] = new DataT[p->length];
                  cnt[s][d][t] = 0;                              // reset
the counter
                }
                dataPtr[s][d][t][cnt[s][d][t]]       =       p->mData;
// store data
                cnt[s][d][t]++;

                if(cnt[s][d][t] == p->length)                          //
packet is complete
                {
                  start[s][d][t]->mDataPtr = dataPtr[s][d][t];
                  flag[s][d][t] = false;

                  // Answer Null means the source is the master
                  if (p -> sResp == Info::SNULL) {
                      d = d - NB_MASTERS;
                      sPort[d]->putReq(*(start[s][d                    +
NB_MASTERS][t]));
                  }
                  // Otherwise the source is the slave
                  else {
                      mPort[d]->putRsp(*(start[s][d][t]));
                  }
                }
            }

            else
            {

                // Answer Null means the source is the master
                if (p->sResp == Info::SNULL) {
                  d = d - MASTERS;
                  sPort[d]->putReq(*p);
                }
                // Otherwise the source is the slave
                else {
                  mPort[d]->putRsp(*p);
                }
            }
```

```
            delete pack;
          }
        }
      }

      int myDecode(Info &info) {

        int s = -1;
        AddrT a = info.mAddr;
        if(info.length > 1) a = info.mAddrPtr[0];

        // -- first see if we are in an interleaved address range
        //
        for(int i = 0; i < numIL; i++) {
            IlI &r = il[i];

            if((a >= r.a1) && (a < r.a2)) {
              s = r.startSlave + ((a>>r.offsetBits) % r.numSlave);
              break;
            }
        }

        if(s < 0)
        {
            s = decode(a);
        }

        if(s < 0 || s >= MAX_SLAVES) {

            fprintf(stderr, "\n*** Error in channel '%s': \n", name());

            fprintf(stderr, "    addr 0x%08x from "
                  "master %d, tid %d is not a valid address!\n\n",
                  (int) a, info.mConnID, info.mThreadID);

            doneSystemC();
        }

        return s;
      }

      void putReq(Info &info) {    //equivalent to source2fifo_in

        numReq++;

        int s;
        Info* ptr;    // burst thing

        s = myDecode(info) + MASTERS;

        //  assert(s >= 0 && s < MAX_SLAVES);

        int m = info.mConnID;
        int t = info.mThreadID;

        if(m == s)
        {
#ifdef ROC_DEBUG1
            cout << "Slave = master, packet directly sent" << endl;
#endif
```

```
            sPort[s]->putReq(info);
        }

        else if((info.length == 1) || (info.length > 1 && info.mCmd !=
Info::WR))
        {
            output = new packet();
            output -> setSource(m);
            output -> setDestination(s);
            output -> setPayload(&info);
            nodes[m]->fifo_in[s] ->write(output);
        }

        else // burst write from master
        {

            start[m][s][t] = &info;
            for(unsigned int i = 0; i < info.length; i++)
            {
              ptr    =    new    Info    (info.mCmd,    *(info.mAddrPtr),
info.mDataPtr[i], info.mConnID, info.mThreadID);
              ptr->length = info.length;       // just to indicate the
packet is part of a burst transaction
              ptr->sResp = info.sResp;

              output = new packet();
              output -> setSource(m);
              output -> setDestination(s);
              output -> setPayload(ptr);
              nodes[m]->fifo_in[s] ->write(output);

            }
        }

        //debugging
#ifdef ROC_DEBUG1
        cout << "-------------------------------------------------------
----------------------" << endl;
        cout << "putReq (m -> s): "   << m << " -> " << s   <<   ",
Address: " <<  info.mAddr;
        Info::MCmd Cmd  = info.mCmd;
        if (Cmd == Info::RD)  cout << "   RD command"  <<  endl << endl;
        else cout << "    WR command (mData): "   << info.mData <<  endl
<< endl;
        cout << "-------------------------------------------------------
----------------------" << endl;
#endif



      }

      // -- Send response to master
      //
      void putRsp(Info &info) {  //equivalent to source2fifo_in

        int s;

        Info *ptr; // burst thing
        int m = info.mConnID;
        assert(m >= 0 && m < MAX_MASTERS);
```

```
        int t = info.mThreadID;

        // + MASTERS is to remove master/slave interface
        s = myDecode(info) + MASTERS;

        if(m == s)
        {
#ifdef ROC_DEBUG1
            cout << "Slave = master, packet directly sent" << endl;
#endif
            mPort[m]->putRsp(info);
        }

        // burst write or normal
        else if((info.length == 1) || (info.length > 1 && info.mCmd !=
Info::RD))
        {
            output = new packet();
            output -> setSource(s);
            output -> setDestination(m);
            output -> setPayload(&info);
            nodes[s] -> fifo_in[m] ->write(output);
        }

        else  // burst read
        {
            start[s][m][t] = &info;
            for(unsigned int i = 0; i < info.length; i++)
            {

              ptr     =     new     Info(info.mCmd,     *(info.mAddrPtr),
info.mDataPtr[i], info.mConnID, info.mThreadID);
              ptr -> length = info.length;
              ptr->sResp = Info::DVA;
              output = new packet();
              output -> setSource(s);
              output -> setDestination(m);
              output -> setPayload(ptr);
              nodes[s]->fifo_in[m] ->write(output);
            }
        }

#ifdef ROC_DEBUG1
        cout << "-------------------------------------------------------
----------------------" << endl;
        cout << "putRsp (s -> m, data): "  << s << " -> " <<  m;
        Info::MCmd Cmd  = info.mCmd;
        if (Cmd == Info::RD)   cout << "    RD command (sData): "   <<
info.sData << endl << endl;
        else cout << "    WR command (ack): "   <<   /*info.mData << */
endl << endl;
        cout << "-------------------------------------------------------
----------------------" << endl;
#endif
      }

      void bind_clk() {
        core -> clk(clk);
        for(i = 0; i < NB_NODES; i++)
        {
            nodes[i]->clk(clk);
```

```
            banks[i]->clk(clk);
        }
      }

      string setParm(SocGenParm &p) {

        if(p.name == "msbDone") {
            printf("msbDone for ROC\n");
            bind_clk();
            return     SocpChannelBase<AddrT,     DataT,     MAX_MASTERS,
MAX_SLAVES>::setParm(p);
        }

        // -- the ileave parameter triggers new IL structure generation
        //
        if(p.name == "ileave") {
            if(numIL >= MAX_ILEAVE)
            return "too many ileave structures defined";

            IlI &i = il[numIL];

            if(
             (i.a1 >= i.a2)                                    ||
             (i.startSlave < 0)                                ||
             (i.numSlave <= 0)                                 ||
             (i.startSlave + i.numSlave >= MAX_SLAVES)     ||
             (i.offsetBits < 0 )                               ||
             (i.offsetBits >= (int)(sizeof(AddrT)*8))
             )
            return "Bad ileave parameters";

            numIL++;

            return "";
        }

        return      SocpChannelBase<AddrT,      DataT,      MAX_MASTERS,
MAX_SLAVES>::setParm(p);
      }

      string getParm(SocGenParm &p) {

        if(p.name == "numReq") {
            gp(p, "numReq", numReq);
            return "";
        }
        return      SocpChannelBase<AddrT,      DataT,      MAX_MASTERS,
MAX_SLAVES>::getParm(p);
      }

      SC_HAS_PROCESS(SocpFuncROC);

       SocpFuncROC(const   char   *name,   bool   isLast   =   true,   int
fifo_in_size = 1000, int fifo_out_size = 1000)
       : SocpChannelBase<AddrT, DataT, MAX_MASTERS, MAX_SLAVES>(name,
false, "ROC"), clk("clock") {

         cout << "Rotate_name:"  << name << endl;
         numIL = numReq = 0;

       SC_CTHREAD(dynamic_run, clk.pos());
```

```
         core = new circular_connect();
      //          core -> clk(clk);    //Later we could set the latency
passed as a parameter

      // node creation, bank creation
      //bind nodes and banks to channel
         for(i = 0; i < NB_NODES; i++)
         {

           string node_name = sform("node%d", i);
           string bank_name  = sform("bank%d", i);

           nodes[i] = new node(node_name.c_str(), i);
           banks[i] = new bank(bank_name.c_str(), i);

           cout << "node_name:"  << node_name .c_str() << endl;
           cout << "bank_name:"  << bank_name .c_str() << endl;

           // connect node to channel interface
           nodes[i]->pr_port(*core);
           nodes[i]->pw_port(*core);
           nodes[i]->req_port(*core);
           nodes[i]->ack_port(*core);
           //               nodes[i]->clk(clk);      //Later we could set
the latency passed as a parameter

           // connect bank to channel interface
           banks[i]->pr_port(*core);
           banks[i]->pw_port(*core);
           banks[i]->req_port(*core);
           banks[i]->ack_port(*core);
           banks[i]->connected_port(*core);
           //             banks[i]->clk(clk);       //Later we could set
the latency passed as a parameter

         }

       // bind the banks together
         for(i = 0; i < NB_NODES; i++)
         for(j = 0; j < NB_NODES; j++)
           {
             //                                                  banks[i]-
>full_out[j](bank_connection[i][j]);
             //                                        banks[(i+1)%NB_NODES]-
>full_in[j](bank_connection[i][j]);
               nodes[i]->fifo_in[j] = new fifo(fifo_in_size);
               nodes[i]->fifo_out[j] = new fifo(fifo_out_size);

             for(int k = 0; k < MAXTHREAD; k++) {
               start[i][j][k] = NULL;
               flag[i][j][k] = false;
               cnt[i][j][k] = 0;
               dataPtr[i][j][k] = NULL;
             }
           }
       end_module();

       }
};
```

```
// -- define some common configurations for SocMon
//

typedef    SocpFuncROC<AddrT,    DataT,MASTERS+SLAVES,MASTERS,    SLAVES>
FuncROCChannel32;
typedef   perNoc<AddrT,   DataT,   FuncROCChannel32,   MASTERS,   SLAVES>
perNoCRoC;

STEPNP_COMPONENT(FuncROCChannel32);     // 32 bit unsigned addr/data
STEPNP_COMPONENT(perNoCRoC);

class perNoCRoCHelper {
 public:
  perNoCRoCHelper() {
    registerAttachHelper("perNocRoc",    "socp32",    pernocAttach<AddrT,
DataT, perNoCRoC>);
  }
};

static perNoCRoCHelper perNoCRoChelp;
```

ANNEXE B
Programme de test pour les simulations fonctionnelles

Cette section contient le code utilisé pour effectuer les simulations fonctionnelles.

```
//=============================================================================
==
//  A demo platform, with demo slave, channel, and master.
//  protocol) abstraction.
//=============================================================================
===
#include <signal.h>
#include <stream.h>
#include <string>
#include <time.h>
#include "systemc.h"
#include <glob.h>

#include <component/processors/generic/socpMasterBase.h>
#include <component/processors/generic/fastMasterBase.h>
#include <component/memories/simple/simpleMemory.h>

#include "demoMaster.h"

typedef uint AddrT;
typedef int  DataT;

#define C_MAX_MASTERS 8
#define C_MAX_SLAVES  8

const int numMaster = 8;
const int numSlave = 8;
int NUMTHREAD = 32;
int prob = 20;

#ifdef MT_MASTER
typedef  DemoMasterBase<AddrT,  DataT,  SocpMasterBase<AddrT,  DataT>  >
MyMaster;
#elif MULTI_DEF_THREAD
#include "MultiDefinedThread.h"
typedef MultiThreadMaster<AddrT, DataT, SocpMasterBase<AddrT, DataT> >
MyMaster;
#elif MULTI_BURST_THREAD
#include "MultiBurstThread.h"
typedef   MultiThreadBurstMaster<AddrT,   DataT,   SocpMasterBase<AddrT,
DataT> > MyMaster;
#else
typedef  DemoMasterBase<AddrT,  DataT,  FastMasterBase<AddrT,  DataT>  >
MyMaster;
#endif

typedef SimpleMemory <AddrT, DataT> MySlave;

#ifdef HROC // hierarchical ROC
#include <component/interconnect/hierarchicalROC/socpFuncHROC.h>
```

```
//typedef SocpFuncHROC<AddrT, DataT, numMaster, (numMaster + numSlave) /
numMaster> MyBaseChannel;
// AddrT, DataT, NB_NODES, NB_SATELLITES
typedef SocpFuncHROC<AddrT, DataT, 4, 2, numMaster, numMaster>
MyBaseChannel;
sc_clock clk;
#elif BIWAYROC // bi-directionnal ROC
#include <component/interconnect/2waysROC/socpFuncROC.h>
// Use SocpFuncROC: The 3 last parameters can be default
typedef SocpFuncROC<AddrT, DataT, numMaster, numMaster, numMaster,
numMaster, numMaster> MyBaseChannel;
sc_clock clk;
#elif ROC
#include <component/interconnect/classicROC/socpFuncROC.h>
// Use SocpFuncROC: The 3 last parameters can be default
typedef SocpFuncROC<AddrT, DataT, numMaster + numSlave, numMaster,
numMaster, numMaster, numMaster> MyBaseChannel;
sc_clock clk;
#elif HYROC
#include <component/interconnect/HyRoc/socpFuncHyROC.h>
// Use SocpFuncROC: The 3 last parameters can be default
typedef SocpFuncHyROC<AddrT, DataT, 4, 4, 4, numMaster, numMaster,
numMaster> MyBaseChannel;
sc_clock clk;
#elif BROC // ROC using bitmap as request
#include <component/interconnect/classicROC/socpFuncROC.h>
// Use SocpFuncROC: The 3 last parameters can be default
typedef SocpFuncROC<AddrT, DataT, numMaster, numMaster, numMaster,
numMaster, numMaster> MyBaseChannel;
sc_clock clk;
#elif RING // ROC using only one bank (token ring)
#include <component/interconnect/ringROC/socpFuncROC.h>
// Use SocpFuncROC: The 3 last parameters can be default
typedef SocpFuncROC<AddrT, DataT, numMaster, numMaster, numMaster,
numMaster, numMaster> MyBaseChannel;
sc_clock clk;
#elif XBAR
#include <component/interconnect/funcXBar/funcXBarChannel.h>
typedef FuncXBarChannel<AddrT, DataT, C_MAX_MASTERS, C_MAX_SLAVES>
MyBaseChannel;
sc_clock clk;
#elif HOTPO
#include <component/interconnect/HotPotato/socpHotPotato.h>
typedef SocpHotPotato<AddrT, DataT, 2,4,C_MAX_MASTERS, C_MAX_SLAVES>
MyBaseChannel;
sc_clock clk;
#elif BUS
#include <component/interconnect/simpleBus/simpleBus.h>
typedef SimpleBus<AddrT, DataT, MyBaseChannel> MyBaseChannel;

#else
#include <component/interconnect/pureFunctional/socpFuncChannel.h>
typedef SocpFuncChannel<AddrT, DataT, C_MAX_MASTERS, C_MAX_SLAVES>
MyBaseChannel;
#endif

#ifdef ICE
#include <component/interconnect/instrumentation/demoIce.h>
//typedef SocpFuncChannel<AddrT, DataT> MyBaseChannel;
typedef SocpICE<AddrT, DataT, MyBaseChannel> MyChannel;
#else
```

```
#ifdef PERNOC
#include "perNocControllerServer.h"
#include <component/interconnect/instrumentation/perNoc.h>
//typedef SocpFuncChannel<AddrT, DataT> MyBaseChannel;
typedef perNoc<AddrT, DataT, MyBaseChannel, C_MAX_MASTERS, C_MAX_SLAVES>
MyChannel;


#else
//typedef SocpFuncChannel<AddrT, DataT> MyChannel;
typedef MyBaseChannel MyChannel;
#endif
#endif


MyChannel *mc = 0;            // -- for terminate stats

//=========================================================================
===
//  ControlC handler prints out performance.
//=========================================================================
===

double startTime;
bool shuttingDown = false;

void
cruelWorld(int arg) {

    if(shuttingDown) return;
    shuttingDown = true;

    printf("\n\t\t\t\t---- End of simulation ----\n");

    double runTime = (double)(clock() - startTime)/CLOCKS_PER_SEC;


#ifdef XBAR
    double numRW = (mc == 0) ? 0.0 : mc->stats.numReq;
    printf("\nTotal Number of Transactions: %d\n", mc->stats.numReq);
#else
    double numRW = (mc == 0) ? 0.0 : mc->numReq;
    printf("\nTotal Number of Transactions: %d\n", mc->numReq);
#endif

    printf("\n\nExecution Time: %9.9f seconds, %d K read/writes per
sec\n",
        runTime, (int)(numRW/(runTime*1000.0)));

#ifdef ROC
    long cycle = mc->getCycle();
    double buffer_use = mc->getBufferUsed();
    cout << "Number of cycles: " << cycle << endl;
    cout << "Average load on buffers: " << buffer_use << " packets" <<
endl;
#elif BROC
    long cycle = mc->getCycle();
    double buffer_use = mc->getBufferUsed();
    cout << "Number of cycles: " << cycle << endl;
    cout << "Average load on buffers: " << buffer_use << " packets" <<
endl;
```

```
#elif BIWAYROC
    long cycle = mc->getCycle();
    double buffer_use = mc->getBufferUsed();
    cout << "Number of cycles: " << cycle << endl;
    cout << "Average load on buffers: " << buffer_use << " packets" <<
endl;
#elif HROC
    long cycle = mc->getCycle();
    double buffer_use = mc->getBufferUsed();
    cout << "Number of cycles: " << cycle << endl;
    cout << "Average load on buffers: " << buffer_use << " packets" <<
endl;
// #elif RING
//     long cycle = mc->getCycle();
//     double buffer_use = mc->getBufferUsed();
//     cout << "Number of cycles: " << cycle << endl;
//      cout << "Average load on buffers: " << buffer_use << " packets"
<< endl;
#endif
    exit(0);
}

//=========================================================================
===
//  SystemC Main
//=========================================================================
===

int sc_main(int ac, char *av[]) {

      int MINDIST = 0;
      int taInterval = 1;

      int   MAXDIST = numMaster;
      int   ACCURACYDIST = 1;


  if (ac == 1)
    {
      /*********** add to make connection betwee C++ nd Java worlds***/
#ifdef PERNOC

      int port = 1234;
      printf("running SIDL server on port %d...\n", port);
      initSidlServer(port);
#endif
      /*******************          end          of          add
**********************************/
      }
            // -- how often to do time accounting
                              // in the channel.
  if(ac == 2) {
    taInterval = atoi(av[1]);
#ifdef PERNOC
    int port = 1234;
    printf("running SIDL server on port %d...\n", port);
    initSidlServer(port);
#endif

  }
```

```
  if (ac == 3)
    {
      taInterval = atoi(av[1]);
#ifdef PERNOC
      int port = atoi(av[2]);
      printf("running SIDL server on port %d...\n", port);
      initSidlServer(port);
#endif
    }
  if (ac == 4)
    {
      taInterval = atoi(av[1]);
#ifdef PERNOC
      int port = atoi(av[2]);
      printf("running SIDL server on port %d...\n", port);
      initSidlServer(port);
#endif
      NUMTHREAD = atoi(av[3]);
    }

      if(ac == 7)
      {
      taInterval = atoi(av[1]);
#ifdef PERNOC
  int port = atoi(av[2]);
  printf("running SIDL server on port %d...\n", port);
  initSidlServer(port);
#endif
  NUMTHREAD = atoi(av[3]);
      MINDIST = atoi(av[4]);
      MAXDIST = atoi(av[5]);
      ACCURACYDIST = atoi(av[6]);
      }
      if ( ac > 4 & ac < 7)
      {
      cout <<endl<<"wrong argument number"<<endl;
      assert(0);
      }
      assert(NUMTHREAD > 0 && NUMTHREAD <= 32);
      assert(MINDIST < MAXDIST);
      assert(ACCURACYDIST >= 1);

  signal(SIGINT, cruelWorld);       // CTRL-C

  assert(numMaster <= C_MAX_MASTERS);
  assert(numSlave  <= C_MAX_SLAVES);

  assert(numMaster == numSlave);    // -- for this demo config

  int numComponents = numMaster;

  MyMaster *demoMaster[numComponents];
  MySlave  *demoSlave[numComponents];

    printf("running   test   with   %d   slaves,   %d   masters,   %d
threads/master...\n",
        numSlave, numMaster, NUMTHREAD);


    printf("hit control-c to exit the program...\n");
```

```
    // -- define the channel
    //
#ifdef ICE
    MyChannel demoChannel("MyDemoChannel", true);
/*
#ifndef ROC
    demoChannel.setSliceParameters(taInterval, 100*taInterval);
#endif
*/
#elif HOTPO
      MyChannel demoChannel("MyDemoChannel", true);
#elif HYROC
      MyChannel demoChannel("MyDemoChannel");
#else
#ifdef PERNOC
  MyChannel demoChannel("MyDemoChannel", true, MINDIST,          MAXDIST,
ACCURACYDIST);
#ifdef XBAR
// Set Latency to 100 and Jitter to 0
  for (int i = 0; i < numComponents; i++)
      {
      for (int j = 0; j < numComponents; j++)
        {
          demoChannel.setLatency(i, j, 100);
          demoChannel.setJitter(i, j, 50);
        }
      }
#else

#ifndef ROC
#ifndef HROC
#ifndef BROC
#ifndef BIWAYROC
#ifndef RING
#ifndef HYROC
    demoChannel.setSliceParameters(taInterval, 100*taInterval);
#endif
#endif
#endif
#endif
#endif
#endif

#endif

#else
    MyChannel     demoChannel("MyDemoChannel",     true,     taInterval,
100*taInterval);  // taInterval = fifo_in_size
#endif
#endif

    mc = &demoChannel;
#ifdef ROC
    demoChannel.clk(clk);
    demoChannel.bind_clk();
#endif

#ifdef BROC
    demoChannel.clk(clk);
    demoChannel.bind_clk();
#endif
```

```
#ifdef HROC
    demoChannel.clk(clk);
    demoChannel.bind_clk();
#endif

#ifdef BIWAYROC
    demoChannel.clk(clk);
    demoChannel.bind_clk();
#endif

#ifdef RING
    demoChannel.clk(clk);
    demoChannel.bind_clk();
#endif

#ifdef HYROC
    demoChannel.clk(clk);
    demoChannel.bind_clk();
#endif

#ifdef HOTPO
    demoChannel.clk(clk);
    demoChannel.bindHopotato();
#endif

    // -- memory regions
    //
    uint memStartAddr = 0;
    uint memSize    = 0x1000;
    uint memEnd;
    int masterID = 0, slaveID = 0;
    for(int i = 0; i < numComponents; i++, memStartAddr += memSize) {

      demoChannel.map(memStartAddr, memStartAddr+memSize, i);
      string masterName = sform("demoMaster%d", i);
      string slaveName  = sform("demoSlave%d", i);
      memEnd = memStartAddr + memSize;

      // -- Make demoMaster i, which checks memory in slave i.
      //
#ifdef MULTI_DEF_THREAD
      // the master can write/read in any slave
/*    demoMaster[i] = new MyMaster(masterName.c_str(), i,
                          memStartAddr, numComponents*memSize); */
      demoMaster[i] = new MyMaster(masterName.c_str(), i,
                          memStartAddr, numComponents, memSize);

#elif MULTI_BURST_THREAD
      int prob = 50;
      demoMaster[i] = new MyMaster(masterName.c_str(), i,
                          memStartAddr,    numComponents,    memSize,
prob);

#else
      demoMaster[i] = new MyMaster(masterName.c_str(), i,
                          memStartAddr, memEnd);
#endif
      MyMaster &m = *demoMaster[i];

      demoChannel.attachMaster(m, m.slavePort, masterID++);
```

```
      // -- make a demo slave
      //
      demoSlave[i]   =   new   MySlave(slaveName.c_str(),   memStartAddr,
memEnd);

      MySlave &s = *demoSlave[i];

      demoChannel.attachSlave(s, s.masterPort, memStartAddr,
          memEnd, slaveID++);
    }

    demoChannel.msbDone();

    startTime = clock();

    sc_start(-1);

    return 0;
}
```

ANNEXE C
Fichier de configuration pour MPEG4

Cette section contient le code du script Python utilisé pour simuler le RoC avec l'application MPEG4. Ce fichier de configuration correspond aux caractéristiques énumérées dans le Tableau 5.6.

```
#!/usr/bin/python
# -*- mode: python -*-
#=====================================================================
======
#  Make a SMP configuration.  This one has 1 ARM running 1 thread at
# 'normal' speed, and the rest running very fast (1000 GHz!) to model
# execution on inf fast hardware.
#=====================================================================
======

import getopt
import sys
import os

if not os.environ.has_key('STEPNP'):
    print """You must have STEPNP environment variable, pointing to
where you installed STEPNP"""
    sys.exit(1);

if not os.environ.has_key('DSOC'):
    print """You must have DSOC environment variable, pointing to
where you installed DSOC"""
    sys.exit(1);

if not os.environ.has_key('SIDL_PORT'):
    sidl_port = -1
else:
    sidl_port = int(os.environ['SIDL_PORT']);

stepNP = os.environ['STEPNP'];
dsoc     = os.environ['DSOC'];

sidl_host = 'localhost'

if os.environ.has_key('SIDL_HOST'):
    sidl_host = os.environ['SIDL_HOST'];

sys.path.append(stepNP + "/lib/python")
sys.path.append(stepNP + "/lib/python/interfaces")

#=====================================================================
===
#  StepNP Imports
```

```
#=============================================================================
===

from sidl   import *;
from sysc   import *;
from cpu    import *;
from socgen       import *;

#=============================================================================
===
#  Option processing
#=============================================================================
===

# -- default values
#
cpus            = 8
threads           = 4
app             = os.path.abspath(sys.path[0]) + '/mpeg4c_socpa_ARM'
debug           = 0
makeSWComponents = 0
dctLatency        = 0
sadLatency        = 0
pernoc            = 0
extpernoc         = 0
nsReceiveRate     = 200000000
filename     = "/mnt/disk_d/superGreg/movie/test.avi"
makeVideoIn  = 1
numThreads   = 32;
interleave        = 0
stbus             = 0
roc               = 0
biroc             = 0
hroc              = 0
hyroc             = 0
numBank           = 1
dcache            = 0
# -- show script usage
#
def usage():
    print"""
Usage:

./socgen [options]

where options are:

    --help                          # prints out this message
    -h              <host>          # host running the SystemC model (def
localhost)
    -p              <port>          # SIDL port number (default SIDL_PORT
env var)
    --g                         # enable debugging options
    --app            <app>      #     application     to     run     (def
mpeg4c_socpa_ARM)
    --cpus              <numCpus>       # number of cpus to enable in
simulation (def 8)
    --threads       <numThreads>  # number of threads per cpu to enable
(def 1)
    --latency       <latency>     # latency for hardware components
    --dctlatency    <latency>     # latency for dct hardware components
```

```
    --sadlatency   <latency>     # latency for sad hardware components
    --r_rate       <r_rate>      # receive rate of the frames in ns
    --sw                         # make only software components
    --file          <filename>     # specifies the avi file to open for
encoding
    --no_videoin         # don't make the hardware VideoIn
    --interleave   <interleaver> # interleaved memory banks
    --numBank      <numBank>     # number of memory bank
    --numThreads   <numThreads>  # number of hardware threads
    --stbus                      # Use STBus as internal channel
    --roc                        # Use RoC as internal channel
    --biroc                         # Use bidirectional RoC as internal
channel
    --hroc                           # Use hierarchical RoC as internal
channel
    --hyroc                        # Use hyper ring-on-chip as internal
channel
    --pernoc                     # Use pernoc on the internal channel
    --extpernoc                  # Use pernoc on the external channel
"""
    sys.exit()

# -- now get command line options
#
try:
    opts, args = getopt.getopt(sys.argv[1:], 'p:h:', ['help',
        'app=',   'cpus=',   'g',   'threads=',   'sw',   'dctlatency=',
'sadlatency=', 'latency=',
        'pernoc',   'r_rate=',   'file=',   'no_videoin',   'numThreads=',
'extpernoc', 'stbus',
        'roc', 'biroc', 'hroc', 'hyroc', 'interleave', 'numBank=']
    )

except getopt.GetoptError:
    print "\nOPTION ERROR"
    usage()

for o, a in opts:
    if o == '-h':
        sidl_host = a
        print "Host is: " + sidl_host
    elif o == '-p':
        sidl_port = int(a)
        print "Port is: " + `sidl_port`
    elif o == '--app':
        app = a
        print "Application is: " + app
    elif o == '--cpus':
        cpus = int(a)
        print "CPUS is: " + `cpus`
    elif o == '--g':
        debug = 1;
        print 'debugging enabled'
    elif o == '--threads':
        threads = int(a)
        print "Threads is: " + `threads`
    elif o == '--help':
        usage()
    elif o == '--sw':
        makeSWComponents = 1
        print "Making SW components"
```

```
    elif o == '--dctlatency' :
        dctLatency = int(a)
        print "Setting dct component latency to %d" % dctLatency
    elif o == '--sadlatency' :
        sadLatency = int(a)
        print "Setting sad component latency to %d" % sadLatency
    elif o == '--latency' :
        sadLatency = int(a)
        dctLatency = sadLatency
        print "Setting hw component latency to %d" % dctLatency
    elif o == '--pernoc' :
        pernoc = 1;
        print "PerNoc Enabled"
    elif o == '--extpernoc' :
        extpernoc = 1;
        print "PerNoc Enabled on external channel"
    elif o == '--r_rate' :
        nsReceiveRate = int(a)
        print "Setting receive rate to %d" % nsReceiveRate
    elif o == '--file' :
        filename = a
        print "Openning file named : " + filename
    elif o == '--no_videoin' :
        makeVideoIn = 0
      app = app + '_no_vi'
        print "Don't make the VideoIn, setting app to : " + app
    elif o == '--numThreads' :
      numThreads = int(a);
      print "Setting HW component numthreds to %d" % numThreads
    elif o == '--stbus' :
      stbus = 1
      print "Using the STBus as internal channel"
    elif o == '--roc' :
      roc = 1
      print "Using the RoC as internal channel"
    elif o == '--biroc' :
      biroc = 1
      print "Using the bidirectional RoC as internal channel"
    elif o == '--hroc' :
      hroc = 1
      print "Using the hierarchical RoC as internal channel"
    elif o == '--hyroc' :
      hyroc = 1
      print "Using the hierarchical RoC as internal channel"
    elif o == '--interleave' :
      interleave = 1
      print "interleaving is on"
    elif o == '--numBank' :
      numBank = int(a);
      print "Setting numBank to %d" % numBank

if sidl_port < 0:
    print "\n\nno port specified (say -p <your port>, or setenv
SIDL_PORT)\n\n"
    usage()

sidlClientMgr.tryConnect(sidl_host, sidl_port);
from sgHelper import *;

#=============================================================================
===
```

```
#  Stuff to dynamic load libs we need
#=============================================================================
===

def dl(lib):
    ldResult = sg.dynamicLoad(lib)
    if ldResult != "":
        print "Problems dynamic loading library '%s':\n    %s" % (
            lib, ldResult
        )
        sys.exit(1);

# -- load base DSOC libraries into stepNP simulator
#
dl(dsoc + '/sc/xsc.so')

# -- Load PerNoC
#
dl(stepNP + "/lib/sidl/perNocControllerServer.so");

# -- load MPEG coprocessors
#
dl(os.path.abspath(sys.path[0]) + '/component/appdom.so')


#=============================================================================
===
#  Boiler plate
#=============================================================================
===

print "\nObjects available for dynamic configuration:";
print "=========================================";
listComponentTypes()
print "\n\n";

#=============================================================================
===
#  top-level net list
#=============================================================================
===

# -- SocGen address map & other parameters
#

# -- Stack management
stackPerThread          = 4096                              # == 16Kb
stackPerProcessor       = stackPerThread*threads


# -- External memory address space
#
RAM_SIZE                = 160                  # MBytes
STACK_SIZE              = stackPerProcessor*cpus
SRAM_PROG_BEGIN         = 0
TEXT_END                = 0x6000
SRAM_PROG_END           = RAM_SIZE*1024*256

# -- Frame buffer address space
#
FRAME_BUFFER_SIZE       = 4096                  # KBytes
```

```
FRAME_BUFFER_START      = SRAM_PROG_END
FRAME_BUFFER_END        = FRAME_BUFFER_START+FRAME_BUFFER_SIZE*256
SRAM_END                = FRAME_BUFFER_END+STACK_SIZE

# -- Local memory size
#
LOCAL_RAM_SIZE          = (SRAM_END)/cpus

SEM_MGR_START           = 0x11000000
SEM_MGR_END             = 0x21000000

HORBA_ENGINE_START      = 0x21000000
HORBA_ENGINE_END        = 0x22000000

PS_CLOCK_PERIOD         = 5000                    # == 200 MHz

masterID          = 0
extMasterID             = 0
slaveID                 = 0

# -- stack management
#
sp                      = SRAM_END

slaves = []
masters = []

#=========================================================================
======
# the external channel
#=========================================================================
======
if( not extpernoc ):
    makeComponent("FuncXBarChannel32",      "extChannel",
                  "enableTimingAnnotation" , '1'      );
else:
    makeComponent("MyPerNoc32", "extChannel");



# -- the EXTERNAL system memory
#
makeComponent('SimpleMemory32',     'systemSRAM',
    'memStartAddr',           `SRAM_PROG_BEGIN`,
    'memEndAddr',             `SRAM_END`,
    'allocate',               '1'
)

attachSlaveToChannel(0, 'systemSRAM', 'extChannel',
    SRAM_PROG_BEGIN, SRAM_END
)


# -- the NUMA architecture memory
#


for c in range(0, cpus):
   l2Name = 'stackRAM' + `c`
   makeComponent('SimpleL2_32'      , l2Name,
                 'cacheSize'        , stackPerProcessor,
```

```
                    'mConnID'          , extMasterID,
                  'accessTimeInNS'   , 0);

   slaves.append((slaveID, l2Name, 'channel',
                    sp-stackPerProcessor, sp))
   attachMasterToChannel(extMasterID, l2Name, 'extChannel')
   sp -= stackPerProcessor;

   slaveID += 1
   extMasterID += 1


l2Name = 'textRAM'
makeComponent('SimpleL2_32'    , l2Name,
              'cacheSize'        , 0x60000,
            'mConnID'          , extMasterID,
              'accessTimeInNS'   , 0);

slaves.append((slaveID, l2Name, 'channel',
                    SRAM_PROG_BEGIN, TEXT_END))
attachMasterToChannel(extMasterID, l2Name, 'extChannel')

slaveID += 1
extMasterID += 1


l2Name = 'heapRAM'
makeComponent('SimpleL2_32'    , l2Name,
              'cacheSize'        , LOCAL_RAM_SIZE*cpus ,
              'mConnID'        , extMasterID,
              'accessTimeInNS'   , 0);
slaves.append((slaveID, l2Name, 'channel', TEXT_END, FRAME_BUFFER_END))
attachMasterToChannel(extMasterID, l2Name, 'extChannel')
slaveID += 1
extMasterID += 1


#=====================================================================
======
# the concurrency engine
#=====================================================================
======

sp = SRAM_END
makeComponent('ConcurrencyEngine32',      'cEngine',
    'memStartAddr',           SEM_MGR_START,
    'memEndAddr',       SEM_MGR_END,
    'accessTime',       0
)

slaves.append((slaveID, 'cEngine', 'channel',
    SEM_MGR_START, SEM_MGR_END
))

slaveID += 1

#=====================================================================
======
#  the HORBA engine
#=====================================================================
======
```

```
makeComponent('HorbaSlave32',    'horbaEngine')

slaves.append((slaveID, 'horbaEngine', 'channel',
    HORBA_ENGINE_START, HORBA_ENGINE_END
))

slaveID += 1


#=====================================================================
======
# the processors
#=====================================================================
======

for c in range(0, cpus):
    armName = 'smpArm' + `c`

    initExternalMem = 0

    if(c == 0):
      initExternalMem = 1

    # -- The ARM processor
    #
    cpuPeriod = PS_CLOCK_PERIOD
    makeComponent('MPEG4_ARM_MT',   armName,

        'heapEnd',                        FRAME_BUFFER_END,
      'TCacheBridgeDevice32',       1,
      'MagicBridgeDevice32',        1,
      'MsgBridgeDevice32',          1,
      'WireBridgeDevice32',         1,
        'ArmCoproDevice32',         1,

      'modelID',              0x12340001,
      'mConnID',              masterID,

        'stackEnd',                       sp,
        'stackPerThread',                 stackPerThread,
        'stackBegin',                     sp - stackPerProcessor,

      'pipeDepth',                  1,
      'numThreads',                 threads,
      'initExternalMem',            initExternalMem,

      'clockPeriod',                   cpuPeriod,
      'clockUnits',                    'SC_PS',

      'numInstrumentRegs',          4,
      'logInstrument',        1,
      'breakInstrument',            0,

      'space',                'mainArmProgram',
      'app',                        app,
      'objDumpProgram',       stepNP + '/bin/arm-elf-objdump',

      'initSTH',                    1,

      'load',                 app
```

```
    )

    if(debug): setParms(armName,
        'proSymbolEnabled',         1,
        'initMHP',                  1)
    if(threads > 1): setParms(armName, 'pipeDepth', 4)

    setParms(armName, 'initCPU', 1);
    if(dcache == 1):
        setParms(armName, 'NWCacheBridgeDevice32_L1', 'dcache');
        setParms(armName+'.dcache.isShared',           0)
        setParms(armName+'.dcache.isWriteBack',        0)
        setParms(armName+'.dcache.Ways',               4)
        setParms(armName+'.dcache.Sets',              64)
        setParms(armName+'.dcache.a1Range',     TEXT_END)
        setParms(armName+'.dcache.a2Range',     SRAM_END)
        setParms(armName+'.dcache.range',              1)
        setParms(armName+'.dcache.releaseA1',   SRAM_END)
        setParms(armName+'.dcache.releaseA2',0xffffffff)
        setParms(armName+ '.dcache.release',           1)


    sp -= stackPerProcessor

    masters.append((masterID, armName, "channel"))

    # -- configure a debugger for this arm
    #
    a = AccessCpuClient(armName);

    a.symbolicDebug('ARM',
        stepNP + '/bin/tgdb',
        os.path.abspath(app),
        "target sim\nload\nrun\n");

    masterID += 1




#=====================================================================
======
#  the DSOC DNS server running in SystemC space
#=====================================================================
======

makeComponent(
    'DemoDNS32',              'myDns',
    'mConnID',                   `masterID`,
    'domNameServer',             'domNameServer',
)

masters.append((masterID, 'myDns', 'channel'))

masterID += 1


if( not makeSWComponents ) :
    makeComponent(
        'MPEGComponent',             'myDCTComponent',
```

```
        'mConnID',                  `masterID`,
      'numThreads',               `numThreads`,
        'dctServer',                'HWDCT',
      'latency',                  `dctLatency`,
    )

    masters.append((masterID, 'myDCTComponent', 'channel'))
    masterID += 1

    makeComponent(
        'MPEGComponent',            'mySADComponent',
        'mConnID',                  `masterID`,
      'numThreads',               `numThreads`,
        'sadServer',                'HWSAD',
        'latency',                  `sadLatency`,
    )

    masters.append((masterID, 'mySADComponent', 'channel'))
    masterID += 1

    makeComponent(
        'MPEGComponent',            'myQuantizeComponent',
        'mConnID',                  `masterID`,
      'numThreads',               `numThreads`,
        'quantizeServer',           'HWQUANTIZE',
    )

    masters.append((masterID, 'myQuantizeComponent', 'channel'))
    masterID += 1


    makeComponent(
        'MPEGComponent',            'myVideoOutComponent',
        'mConnID',                  `masterID`,
      'numThreads',               `numThreads`,
        'videoOut',               'VIDEOOUT',
      'latency',                  '0',
    )

    masters.append((masterID, 'myVideoOutComponent', 'channel'))
    masterID += 1
    if(makeVideoIn):
        makeComponent(
            'MPEGComponent',            'myVideoInComponent',
            'mConnID',                  `masterID`,
            'bufStart',                 `FRAME_BUFFER_START`,
            'bufEnd',                   `FRAME_BUFFER_END`,
            'receive_rate',             `nsReceiveRate`,
            'filename',                 filename,
            'rxFrame',                  'rxFrame',
            'handleFrame',              'handleFrame',
        )

        masters.append((masterID, 'myVideoInComponent', 'channel'))
        masterID += 1

#=====================================================================
======
# the internal channel
#=====================================================================
======
```

```
if(stbus):
    makeComponent('SGClock', 'systemClock',
        'onTime', PS_CLOCK_PERIOD/2,
        'offTime', PS_CLOCK_PERIOD/2,
        'units', 'SC_PS')

    makeComponent('SGReset', 'systemReset',
        'resetTimeNS', 10000)

    if not pernoc:
        makeComponent('BaseSTBus', 'channel')
#        makeComponent('hPSTBT', 'channel')
    else:
        makeComponent('PerNocSTBus', 'channel')
#        makeComponent('perNoChPSTBT', 'channel')


    attachComponent('systemClock', 'clock', 'channel', 'channel.clock');
    attachComponent('systemReset', 'reset', 'channel', 'channel.reset');

elif(roc):
    makeComponent('SGClock', 'systemClock',
                  'onTime', PS_CLOCK_PERIOD/2,
                  'offTime', PS_CLOCK_PERIOD/2,
                  'units', 'SC_PS')

#      setParms('systemClock', 'onTime', PS_CLOCK_PERIOD/2, 'offTime',
PS_CLOCK_PERIOD/2,'units', 'SC_PS')

    if not pernoc:
        makeComponent('FuncROCChannel32', 'channel')
    else:
        makeComponent('perNoCRoC', 'channel')

    attachComponent('systemClock', 'clock', 'channel', 'channel.clock');

elif(biroc):
    makeComponent('SGClock', 'systemClock',
                  'onTime', PS_CLOCK_PERIOD/2,
                  'offTime', PS_CLOCK_PERIOD/2,
                  'units', 'SC_PS')

#      setParms('systemClock', 'onTime', PS_CLOCK_PERIOD/2, 'offTime',
PS_CLOCK_PERIOD/2,'units', 'SC_PS')

    if not pernoc:
        makeComponent('Func2WayROCChannel32', 'channel')
    else:
        makeComponent('perNoC2WayRoC', 'channel')

    attachComponent('systemClock', 'clock', 'channel', 'channel.clock');

elif(hroc):
    makeComponent('SGClock', 'systemClock',
                  'onTime', PS_CLOCK_PERIOD/2,
                  'offTime', PS_CLOCK_PERIOD/2,
                  'units', 'SC_PS')

#      setParms('systemClock', 'onTime', PS_CLOCK_PERIOD/2, 'offTime',
PS_CLOCK_PERIOD/2,'units', 'SC_PS')
```

```
    if not pernoc:
        makeComponent('FuncHROCChannel32', 'channel')
    else:
        makeComponent('perNoCHRoC', 'channel')

    attachComponent('systemClock', 'clock', 'channel', 'channel.clock');

elif(hyroc):
    makeComponent('SGClock', 'systemClock',
                  'onTime', PS_CLOCK_PERIOD/2,
                  'offTime', PS_CLOCK_PERIOD/2,
                  'units', 'SC_PS')

#       setParms('systemClock', 'onTime', PS_CLOCK_PERIOD/2, 'offTime',
PS_CLOCK_PERIOD/2,'units', 'SC_PS')

    if not pernoc:
        makeComponent('FuncHYROCChannel32', 'channel')
    else:
        makeComponent('perNoCHyRoC', 'channel')

    attachComponent('systemClock', 'clock', 'channel', 'channel.clock');

else:
    if( not pernoc ):
        makeComponent("FuncXBarChannel32",      "channel",
                      "enableTimingAnnotation" , '1'      );
    else:
        makeComponent("MyPerNoc32", "channel");
    setParms('channel',
             'baseLatency', '0',
             'baseJitter', '0')

print "Masters: %d - Slaves: %d" % (masterID,slaveID)

map(lambda m: attachMasterToChannel(m[0], m[1], m[2]), masters)
map(lambda  s:  attachSlaveToChannel(s[0],  s[1],  s[2],  s[3],  s[4]),
slaves)


#=====================================================================
======
#  Indicate channel binding done, and start the simulation
#=====================================================================
======
setParms('channel', 'msbDone', '1')
setParms('extChannel', 'msbDone', '1',
         'baseLatency', '0',
         'baseJitter', '0')



print "\nObjects Configured";
print   "==================";
listComponents()

sg.startSystemC(-1);

print "Masters: %d - Slaves: %d" % (masterID,slaveID)
print "\n\nModeled configured, ready to start simulation..."
```

```
# -- do this from startUp script...
#
#systemC = SystemC_Client("SystemC_Server", "access0");
#systemC.runModel("cycle", '10000000000');
```

Code de multi_defined_thread ?

Code de funcTest?

script socgen

ANNEXE D
Article soumis à la conférence ICCD 2005

Cette section présente l'article découlant des travaux de recherche effectués. Cet article a été soumis à la conférence ICCD. Au moment de la rédaction de ces lignes, l'article était soumis à la révision par le comité de sélection.

RoC: A Scalable Network on Chip Based on the Token Ring Concept

François Deslauriers[1], Michel Langevin[2], Guy Bois[1], Yvon Savaria[1], Pierre Paulin[2]

[1] École Polytechnique de Montréal, QC, Canada
{deslaur, bois, savaria } @ grm.polymtl.ca

[2] STMicroelectronics, ON, Canada
{michel.langevin, pierre.paulin } @ st.com

Abstract

A recent practice in the development of SoCs is the integration of interconnect networks, since integration offers significant bandwidth increases. This allows implementing multiprocessors systems that communicate more effectively than bus based architectures. This paper proposes a new network-on-chip based on the token ring model. This easily scalable network has been integrated into a system level exploration platform for characterization. Increased performance is confirmed and improvements are proposed to decrease packet latency through the network.

1. Introduction

Technology scaling will allow Systems-on-Chips (SoCs) containing hundreds of complex cores in the near future [1]. This will lead to new applications in fields like telecommunication and entertainment [2]. A promising approach to design flexible high performance SoC architectures is to organize them as multiprocessors, where the ability for processors to communicate effectively with each other is critical. Tomorrow's on-chip bandwidth requirements will easily reach hundreds of Gbit/s. Nowadays, buses are widely used, but they are increasingly becoming system bottlenecks as their bandwidth is usually shared between several modules. Buses can be segmented with bridges to support multiple simultaneous data transfers, and they can also be organized according to hierarchical structures. Nevertheless, buses are hard to scale. A significant challenge with future highly parallel SoC architectures is to avoid communications being their system bottleneck.

Networks on Chip (NoCs) have been proposed as a means to address these issues. NoCs attempt to meet the requirements for future SoCs with respect to reusability, scalability and parallelism.

In this paper, we introduce the RoC (*Rotator on Chip*), a scalable network based on the token ring concept. RoC guarantees packet arrival order and is fairly simple to implement.

The following section presents an overview of existing NoC architectures. Section 3 presents guiding principles behind the RoC concept, followed in section 4 by optimizations and features added to the architecture. Section 5 presents performance analysis results while Section 6 presents final remarks about RoC and suggests directions for future research.

2. Networks on chip

NoCs typically exploit a packet passing communication model, as in larger scale networks like LANs or the WWW. They exploit the same concepts as large scale networks:

- messages are decomposed into packets with header, payload and trailer;
- packets are routed from source to destination according to a specified algorithm;
- processing cores require wrappers (also called network interfaces) to be attached to the network terminals, in order to support different protocols and communication models, and
- various strategies are used for routing, data control, switching, buffering, and arbitration.

2.1 SoC interconnection topologies

Popular topologies for NoCs include crossbars, trees and meshes. The crossbar is known for its low latency, high cost and poor scalability. Tree topologies [4], [5] offer a good latency, but they are coupled to very high wiring costs. Moreover, in fat tree topologies, routers are quite costly in area (a partial crossbar). Although wormhole routing allows a bandwidth

increase, contention between message pairs may block several links at the same time. Adaptive routing improves the saturation threshold, but its use imposes that packets must be reassembled at the receiving end, since packet arrival order may be random, particularly at high loads. Mesh topologies [2], [6] offer a large bandwidth, but are non-deterministic, and their latency can be very high under heavy traffic. In some cases, packets are not guaranteed to reach destination, since a high traffic can lead to livelock.

Some networks are already on the market. For instance, ClearConnect® [7] provides a good bandwidth by replacing a typical bus structure with a chain of switches, allowing several simultaneous transfers on different chain segments. Also, STBus [8] is a modular architecture using several nodes in different configurations to obtain a hierarchical structure. A node is considered a partial crossbar as it can go from a shared bus to a full crossbar, depending on the configuration. The RoC emerged from the need for a less costly NoC that offers good performance.

3. The RoC System Architecture

The RoC project is part of a broader research on SoC design methodologies. As part of this research, it was observed that communications represent a system bottleneck, and that using low latency interconnect network is imperative. RoC is a novel NOC architecture based on the general token ring network model. It addresses the requirements for scalable and simple integrated networks.

The architecture of the RoC is illustrated in Figure 1 and its modules are detailed in the following sections.

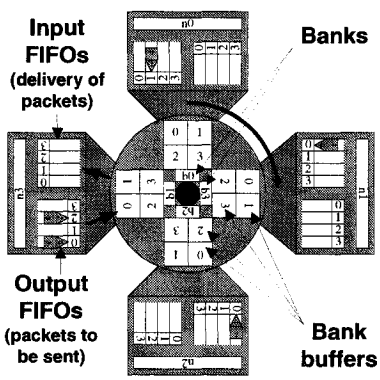


Figure 1 : RoC high level block diagram with 4 resources.

3.1 Node interface

In the following discussion, it is assumed that we want to connect together N resources (N is equal to 4 in the example of Figure 1). For instance, a resource can be a processor, a coprocessor, a memory, a dedicated-hardware component, a bridge used to connect to another communication channel or any data processing device. Each resource must be attached to a node interface (also called network interface – **n0** to **n3** in Figure 1). Each node is composed of N - 1 input FIFOs and N - 1 output FIFOs (one per destination). The fixed-size packets to be sent are tagged with source and destination ID, and queued in associated FIFOs.

3.2 Core-switch

The core-switch is composed of N memory banks (**b0** to **b3** in Figure 1), each bank being composed of N bank buffers. Each buffer is associated to a node ID, such that the tag is used as the destination address. Since each bank contains N buffers, the number of required buffers is N^2 (means to reduce this number are proposed in Section 4). The core-switch turns (i.e. changes its connections to the nodes) clockwise and makes one full rotation in N **steps** (the description of a step is given below). At each step, each bank is connected with exactly one node. A node can send one packet to the connected bank only if the buffer associated with the packet destination is empty. Each node may send at most one packet per step. Also, during the same step, a bank can deliver a packet to the connected node to which it is associated. It means that a packet sent from one node to another requires at least two steps to get through (one to send the packet from the input node to a bank and one to send it from the bank to the output node).

Every step, each node examines the status of its output FIFOs and builds a transfer request according to a given priority scheme. The classic RoC uses a round-robin algorithm because of its simplicity, although more complex priority-based algorithm could be used. The request contains information about the packet source and its destination. If the request is accepted, the node will send the packet during the next step (i.e. when the core will have turned one step). If not, no transfer will occur. Therefore, a packet sent by a node follows an acknowledgement received a step earlier, i.e. a

packet is sent to the connected bank (bank #I), while the node initiates a request for the next transfer (bank #I+1). The behavior for one step is summarized in Figure 1. Note that this process involves two priority schemes: the FIFO selection done at the node and the request acceptance/rejection done by the bank controller.

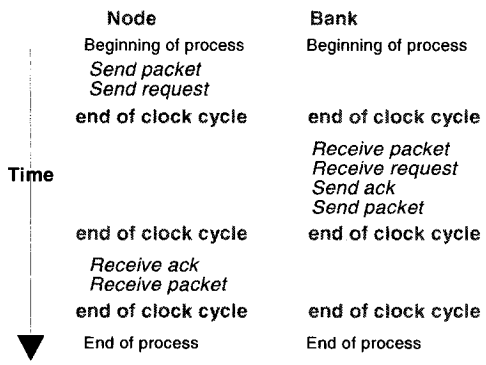


Figure 1 : Functional behavior of components during a step

The RoC model was built using SystemC 2.0 [9] and integrated in StepNP (System-Level Exploration Platform for Network Processors). StepNP is a SystemC environment used to explore applications and architectures [3]. It easily allows adding processors, coprocessors, memories, and interconnects fabrics. Modules that compose the platform communicate using the OCP protocol [10]. That means the fixed-sized RoC packet is composed of three fields as the OCP packet is wrapped with a source ID and a destination ID, as shown in Figure 2. An example of how many bits are required by every field is also provided in Figure 2. Depending on the communication protocol used between resources, packet size may vary from 50 bits up to 200 bits.

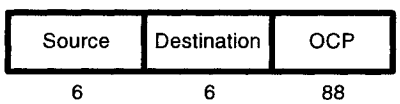


Figure 2 : Representative structure of a RoC packet

The RoC guarantees packet arrival order, since the core-switch is token ring like. Also, a connection from A to B cannot block a connection from C to D, for any C and D different from A and B, making the RoC internally non-blocking for packets with different sources and destinations.

4. Proposed Improvements

In the following sections, improvements to ROC are proposed. The gains achieved by those improvements are quantified in Section 5.

4.1 Buffer optimization

The basic model of the RoC (Section 3) considers N buffers in each bank. This is very costly in area and consumes power. Also, simulations showed that even under heavy traffic, the buffer occupancy rate is around 50% (see Section 5). Indeed, considering a distributed traffic, packets travel half the nodes before getting to destination, on average. Then, we could expect that on average half the buffers of a given bank will be free at any given time. Therefore, removing half the buffers allows preserving performance while reducing significantly the power consumed by the NOC. This reduces implementation complexity and may allow adding extra nodes. However, arbitration algorithm must be adapted. This issue is briefly discussed in Section 5.2.

4.2 Bitmap request

As mentioned earlier, a bank sends back a NACK (not-acknowledged) to a node if the next bank's buffer associated with the destination is full. For instance, let node #1 be connected to bank #0 in a given step. Node #1 sends a request for a transaction to node #3. Bank #0 then examines buffer #3 of bank #1 (the next one in the rotation). If this buffer is full, bank #0 will send back a NACK to node #1. Thus, node #1 will not send any packet during the next step. However, another buffer (either #0 or #2) could be available and used for a transaction. To solve this problem and to let each node the possibility to transmit a packet at each step, a node can send a bitmap giving every possible destination for ready to transmit packets, rather than sending destination number for a unique packet (Figure 3). In this case, the bank controller examines the request and sends back a buffer number telling the node which packet it must send during the next step. The buffer number can be chosen using a round-robin algorithm. A priority-based algorithm could also be supported, although it would be more costly.

To summarize, in the basic model of the RoC, part of arbitration decisions are executed by the node controller, while the introduction of the bitmap request sends all arbitration decisions to the bank side. The node simply sends a request based on its FIFOs' states.

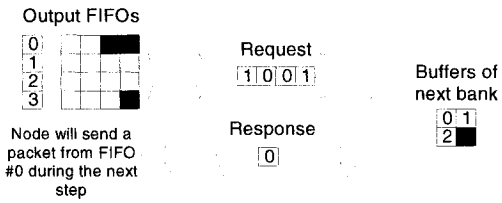


Figure 1 : Bitmap request mechanism

4.3 Bi-directional RoC

A good way to map a high proportion of node pairs communicating heavily would be side by side. This configuration is best for a transaction in the rotation direction, but is the worst for a transaction occurring in the opposite direction, like in the original token ring model. Indeed, even if two nodes are neighbors, packets exchanged in a bi-directional flow between them will still have to travel all the way around.

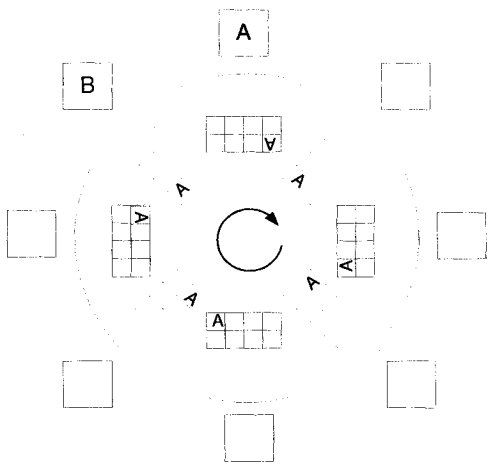


Figure 2 : Contention problem

Also, under heavy traffic, the latency for a rotation-direction transfer between two neighbor nodes becomes a problem. For instance, let us consider an eight-node RoC, where seven masters (processors) request the same slave node (a memory). Let node A (that includes the memory) be located right next to node B (that includes a processor) in the rotation (Figure 2). Each time node B wants to send a packet to node A, a request is sent, as explained earlier. In that case, node B will receive an acknowledgment if buffer #A is empty on the connected bank. However, there is a high probability that the buffer will already contain a packet since the six remaining processors have been connected to this bank before. Even though the transfer between node B and A seems very fast considering the topology, a large amount of time can occur before the transfer begins. This situation is also illustrated in Figure 5 (Section 5.1) where the latency of packets sent between master 0 and slave 7 increases with the number of threads per master.

A good way to alleviate these problems is to use a bi-directional rotator, as shown in Figure 3. The banks are divided in two sets rotating in opposite directions. Thus, each node will be successively connected to a clockwise direction moving bank and a counterclockwise direction moving bank. This solution allows mapping a pair of nodes communicating heavily side to side, since the transaction latency will be reduced to its lowest in both directions. Thus, proper selection of the direction must be included in the priority scheme. Also, under heavy traffic, the overall latency (including the delay due to the request being rejected) between node B (processor) and node A (memory) will decrease because half the nodes to which buffers rotating in a direction will use this direction in priority (the other half will have sent their packet the other way). Thus, the contention probability is lowered and there is a higher probability that the transaction will occur, which reduces average latency.

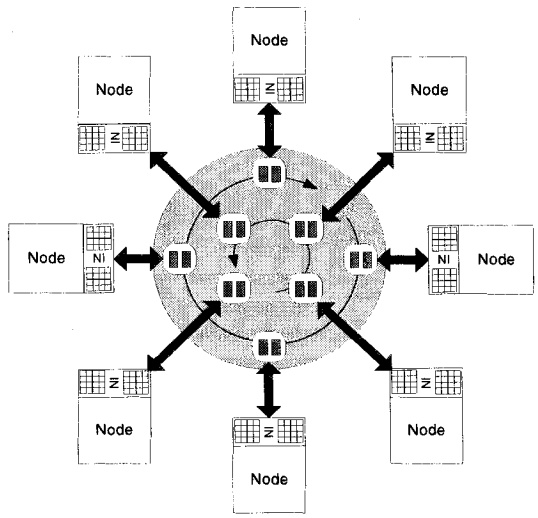


Figure 3 : Bi-directional rotator

4.4 Hierarchical RoC

When the number of interconnected nodes is high, the average latency becomes large, even using a bi-directional rotator. This is the main disadvantage of the token ring topology. Keeping latency at a reasonable level regardless of the number of nodes suggests the use of a hierarchical RoC, as shown in Figure 1.

In the hierarchical structure, a basic rotator is called a satellite, and many satellites are interconnected using a central rotator. In a balanced 2-level hierarchy, each satellite contains N/S nodes, where N is the total number of nodes to connect and S is the number of satellites. Among those nodes, some use a special node interface (bridge) to forward packets from a satellite to another. Those bridges can be viewed as dedicated processors, or processors performing a special kind of transactions.

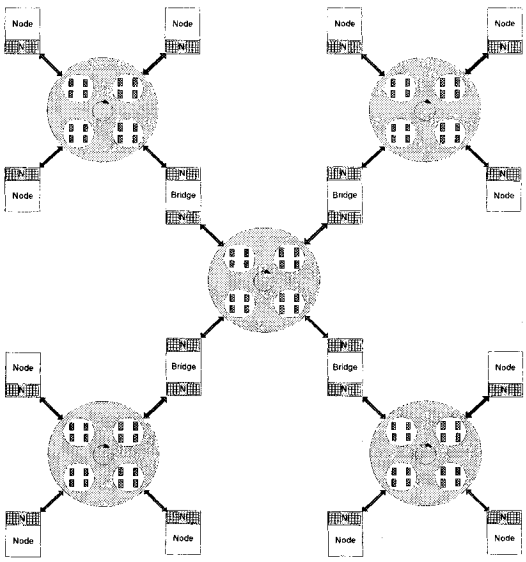


Figure 1 : Hierarchical RoC

The worst case of this topology occurs when a packet requires N/S steps to reach the special interconnect node, S - 1 steps to reach the destination satellite and N/S steps to get to destination. Then, a total of 2(N/S) + S - 1 steps may be required, compared to N – 1 steps in the initial RoC. For example, considering a 16 nodes system with 4 satellites, it requires 11 steps instead of 15. For a 128 nodes system with 8 satellites, the maximum number of steps is 39 instead of 127! This leads to significant latency reduction. Moreover, a clever application mapping would keep most of the transactions local (on a satellite), rather than global. Finally, this configuration makes the RoC less costly in area. Although S^2 additional buffers are needed for the central rotator, all the satellite banks are smaller, because they comprise fewer nodes. For instance, to connect 128 nodes, 128 * 128 = 2^{14} buffers are needed. With 8 satellites supporting 16 nodes each, 8*17*17 + 8*8 = 2376 buffers are needed, which is much less than 2^{14}. The gain is even better as N grows. It shows how the ROC can be easily scaled while remaining affordable.

5. Results

This section presents results of simulations conducted to confirm the RoC functionality and to characterize its performance.

As mentioned previously, the RoC was simulated using the StepNP platform that regroups several IP models, such as processors and memories. An instrumentation tool, named PERNoC (PERformance NoC) and also part of StepNP, probes results produced during simulations and analyzes performance according to specified metrics, such as latency and offered bandwidth.

The simulations involved eight nodes. Each node can be considered as a cluster comprising a multi-threaded processor (the master) and its memory (the slave). Each thread repetitively writes a 32-bit integer somewhere in the memory of other nodes (random destination address) and reads it back. Since the read and write transactions done on respective processors are blocking (i.e. the process is stopped until the transaction has completed), multi-threading is used to increase the load on the network. The multi-threading degree is a configurable parameter specified in respective experiments. Results were taken after 5 million (simulated) clock cycles.

5.1 Load

Figure 2 shows that an 8-node RoC can route up to 64 packets (8 threads on 8 masters) at the same time without observing a significant increase of its latency (note the scale change between graphs). Further increasing the load may lead to an undesirable behavior. Examining the right columns (8 and 12 threads) of Figure 2, we notice that a packet from node #0 to node #7 (his neighbor) takes a long time to get to destination. This behavior has already been discussed in Section 4.3.

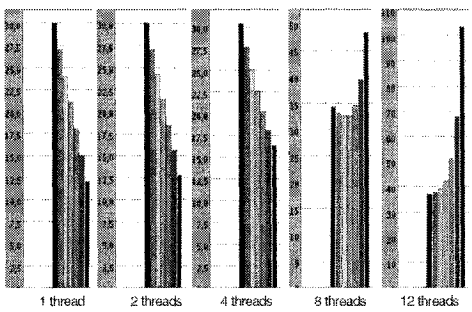


Figure 1 : Average packet latency for a transfer between master 0 and slave 1 to 7 (from left to right resp.), according to the number of threads per master. Rotation occurs in a descending order of destination addresses

An interesting NoC characteristic is the utilization rate of its interface (also called load). Assuming a three-clock rotator step, the utilization can be defined as the **Number of threads * 3 / Average delay in clock cycles**. Figure 2 shows that RoC saturates for an offered load of 66% before latency becomes large and tends to go to infinity with random traffic. This is much better than the 28% saturation rate obtained with the SPIN fat tree network [4].

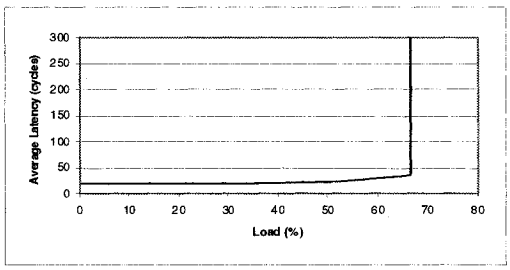


Figure 2 : RoC latency/load

5.2 Buffers

As discussed in Section 4.1, simulations confirmed that half of the buffers could be removed from the banks without affecting performance, because the occupancy rate rarely exceeds 50% (see Table 1). A possible means to do so is proposed later. Moreover, the fraction of removed buffers can grow up to 75% when using a bi-directional RoC, as shown in Figure 3 where a reduction of the number of buffers of 75% does not increase the latency.

Number of threads	2	4	8	12	16
Classic RoC	14.0%	26.4%	32.3%	32.9%	33.0%
Bitmap RoC	14.3%	26.9%	40.4%	43.4%	44.0%
Bi-directional RoC with bitmap	9.3%	16.1%	21.9%	23.3%	23.6%
Hierarchical RoC with bitmap (two 4-node satellites)	7.9%	14.6%	16.8%	16.9%	16.9%

Table 1 : Average buffer occupancy rate with random traffic

The proposed reduction in number of buffers can have a major impact on area and power consumption. However, the arbiter must be a little more intelligent because the algorithm changes slightly: for instance, the bank cannot verify in the next bank if the buffer corresponding to the destination is free, because there is now fewer buffers than the number of destinations. The buffers would now shared, making the algorithm more complex. For instance, with half the buffers removed, an easy way to proceed would be to tag the buffer with the destination number shifted right by one bit. Thus, buffer #0 would be associated with destinations #0 and #1, buffer #1 would be associated with destination #2 and #3, and so on. Despite this change, removing useless buffers is a major improvement.

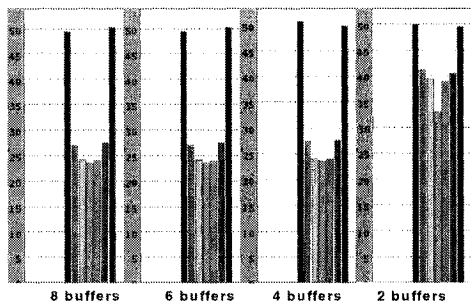


Figure 3 : Packet latency for a transfer between master 0 and slave 1 to 7 (from left to right resp.) using a bi-directional RoC, according to the number of buffers on banks (8 threads/master).

5.3 Global performance

The insertion of additional nodes on the network leads to additional available bandwidth, unlike what can be seen using token ring or bus architectures (see Figure 4) As the bandwidth provided by a basic token ring network capable of one transfer at a time quickly saturates, the bandwidth associated with RoC models continues to grow as more resources are connected to the network. Indeed, the modular RoC structure allows processing multiple

simultaneous transactions. The use of a bitmap request leads to more transfers and thus increases effective bandwidth, as shown in Figure 2, where a 25% increase in the number of completed transactions is observed. In this case, more packets get through the network because available resources are better managed. Also, the bi-directional RoC leads to a lower latency, since packets travel in fewer steps before reaching destination. These two features allow the nodes to use more efficiently the bandwidth offered by the core-switch buffers. Figure 1 also provides information about which configuration can be selected for a given traffic pattern. For instance, with random traffic, the hierarchical structure leads to reduced performance compared to the original model because of additional steps taken to go from a satellite to another. Therefore, a hierarchical ROC is less suitable for a random traffic, but it is more appropriate for clustered traffics.

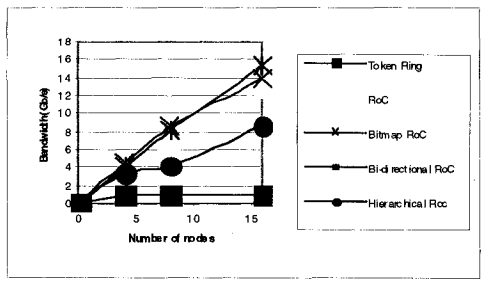


Figure 1 : Effective bandwidth for different RoC configurations

6. Conclusion and future work

This work presented the Rotator on Chip, a parameterized and scalable NoC architecture based on the token ring concept. It was designed to produce a low latency and to maximize the use of the available bandwidth without consuming too much resources. Features such as bi-directional transmission, and bit-map priority resolution can improve its performance. A hierarchical version of RoC was proposed to limit latency as well as means to improve buffer utilization efficiency. It was shown that RoC can support higher utilization rate than SPIN before saturating.

Currently, we are working on additional features that can be added to existing models, such as bandwidth reservation and flow control mechanisms. Also, we are developing an RTL model of the RoC to better assess its characteristics in terms of die area, clock operating frequency, and power consumption.

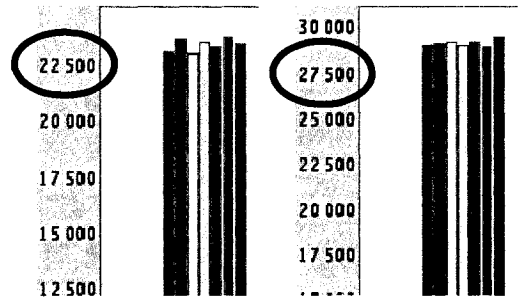


Figure 2 : Number of transactions between master 0 and slave 1 to 7 using a) a standard request form and b) a bitmap request form.

7. References

[1] L. Benini and G. De Micheli, "Networks on chip: A new SOC paradigm", *IEEE Computer*, Jan. 2002, pp.70-78.

[2] C. A. Zeferino and A. A. Susin, "SoCIN: A Parametric and Scalable Network on chip", *SBCCI'2003*, p. 169-174

[3] P. Paulin, C. Pilkington and E. Bensoudane, StepNP: "A System-Level Exploration Platform for Network Processors", *Design & Test of Computers*, June 2002, pp. 17-26.

[4] A. Adriahantenaina, H. Charlery, A. Greiner and L. Mortiez, "SPIN: a Scalable, Packet Switched, On-Chip Micro-network", *DATE'2003*, p.70-73

[5] C. Grecu, P. Pande, A. Ivanov and R. Saleh, "Design of a Switch for network on Chip Applications", IEEE *ISCAS'2003*, p. 217-220

[6] E. Nilsson, "Design and Implementation of a hot-potato Switch in a Network on Chip", Master of Science thesis, June 2002, 66p.

[7] CLEARSPEED™, *ClearConnect® Bus : High Performance On-Chip Interconnect*, interconnect description, 2003.

[8] STMicroelectronics, *STBus Communication System Concepts and Definitions.*, 2003, Specifications are available at http://www.stmcu.com/inchtml-pages-STBus_intro.html

[9] *The Open SystemC™ Initiative*, http://www.systemc.org

[10] *Open Core Protocol Specification Release 1.0*, 202p.